STM32Java
UI Extension

STM32JavaF2 - Keil uVision

User Manual

Reference:	TLT-0614-MAN-STM32JavaF2
Revision:	C
Architecture:	STM32JavaF2
Compiler:	Keil uVision
Product Version:	4.0.3

Confidentiality & Intellectual Property

All right reserved. Information, technical data and tutorials contained in this document are confidential, secret and IS2T S.A. Proprietary under Copyright Law. Without any written permission from IS2T S.A., *copying or sending parts of the document or the entire document by any means to third parties is not permitted* including but not limited to electronic communication, photocopies, mechanical reproduction systems. Granted authorizations for using parts of the document or the entire document do not mean they give public full access rights.

IceTea®, IS2T®, MicroJvm®, MicroEJ®, S3™, SNI™, SOAR®, Drag Emb'Drop™, IceOS®, Shielded Plug™ and all associated logos are trademarks or registered trademarks of IS2T S.A. in France, Europe, United States or others Countries.

Java™ is Sun Microsystems' trademark for a technology for developing application software and deploying it in crossplatform, networked environments. When it is used in this documentation without adding the ™ symbol, it includes implementations of the technology by companies other than Sun.

Java™, all Java-based marks and all related logos are trademarks or registered trademarks of Sun Microsystems Inc, in the United States and other Countries.

Other trademarks are proprietary of their authors.

Table of Contents

1. Bibliography 6
2. Introduction 7
    2.1. Scope 7
    2.2. Intended audience 7
    2.3. Related Documents 7
    2.4. Document Organization 7
3. Platform Design 8
4. MicroUI 11
    4.1. Introduction 11
    4.2. Architecture 11
    4.3. MicroUI Internal 12
    4.4. Threads 12
    4.5. Layers 13
    4.6. Transparency 15
    4.7. Dependencies 16
    4.8. Installation 16
    4.9. Use 16
5. Static Initialization 17
    5.1. Introduction 17
    5.2. Functional Description 17
    5.3. Root Element 18
    5.4. Display Element 18
    5.5. Event Generators Element 18
    5.6. Example 19
    5.7. Dependencies 19
    5.8. Installation 20
    5.9. Use 20
6. LEDs 21
    6.1. Introduction 21
    6.2. Implementations 21
    6.3. Dependencies 21
    6.4. Installation 21
    6.5. Use 21
7. Inputs 22
    7.1. Introduction 22
    7.2. Functional Description 22
    7.3. Implementations 23
    7.4. Dependencies 24
    7.5. Installation 24
    7.6. Use 24
8. Display 25
    8.1. Introduction 25
    8.2. Display Configurations 25
    8.3. Buffer Modes 25
    8.4. Memory Layout 28
    8.5. Pixel Structure 29
    8.6. Implementations 29
    8.7. Low Level API 30
    8.8. Dependencies 32
    8.9. Installation 32
    8.10. Use 33
9. Audio 34
    9.1. Introduction 34
    9.2. Dependencies 34
    9.3. Installation 34

10. Alpha Numeric Display 35
10.1. Introduction 35
10.2. Dependencies 35
10.3. Installation 35
11. Images 36
11.1. Image Engine Core 36
11.2. Image Generator 37
11.3. Image Decoder 41
12. Fonts 44
12.1. Font Engine Core 44
12.2. Font Designer 47
12.3. Font Generator 52
13. MWT 54
13.1. Introduction 54
13.2. Widgets 54
13.3. Transition 54
13.4. Dependencies 54
13.5. Installation 55
13.6. Use 55
14. Story Board 56
14.1. Story Board Engine Core 56
14.2. Story Board Designer 57
14.3. Story Board Generator 60
15. Motion 62
15.1. Introduction 62
15.2. Dependencies 62
15.3. Installation 62
15.4. Use 62
16. Flow 63
16.1. Introduction 63
16.2. Dependencies 63
16.3. Installation 63
16.4. Use 63
17. Simulation 64
17.1. Introduction 64
17.2. Functional Description 64
17.3. The Front Panel Project 64
17.4. FP File 66
17.5. Extensions 68
17.6. Dependencies 70
17.7. Installation 70
17.8. Use 71
18. Launch Options 72
18.1. Category: Libraries 72
18.2. Category: Target 77
19. Document History 80

List of Figures

3.1. The user interface extension components along with a STM32JavaF2 ARMCCv4 platform 8
3.2. Overview 9
4.1. MicroUI Elements 11
4.2. MicroUI APIs 12
4.3. User-defined Event Generator Declaration 12
5.1. MicroUI Process 18
5.2. Root Element 18
5.3. Display Element 18
5.4. Event Generator Element 19
5.5. MicroUI Initialization File Example 19
7.1. Drivers and MicroUI Event Generators Communication 22
7.2. MicroUI Events Framework 23
8.1. Display Direct Mode 28
8.2. Memory layouts 29
8.3. Buffer Modes 30
8.4. Display Driver Example 31
8.5. Display Driver Declaration 32
11.1. Image Engine Core Principle 36
11.2. Image Generator Principle 37
11.3. Image Generator Configuration File Example 38
11.4. Image Generator Extension Project 39
11.5. Image Generator Extension Implementation Example 40
11.6. BITMAP Output Format Examples 40
11.7. Image Decoder Principle 42
11.8. Image Decoders Table 43
12.1. Font Generation 44
12.2. Font height 45
12.3. Font baseline 45
12.4. Default character 45
12.5. Font Generation 47
12.6. Font Height 48
12.7. The Baseline 48
12.8. Character Editor 50
12.9. Font Preview 51
12.10. Font Generator Principle 52
12.11. Fonts Configuration File Example 53
14.1. Story Board Generation 56
14.2. Story Board Generation 57
14.3. Storyboard Designer Application 58
14.4. Page Groups Navigation Example 59
14.5. Story Board Generator Principle 60
17.1. New Front Panel Project Wizard 65
17.2. Project Content 65
17.3. Working Layout Example 67
17.4. Active Area 68
17.5. `.fp` file - display example 68
17.6. `.fp` file - push example 69

List of Tables

4.1. MicroUI C libraries 11
8.1. Switch mode synchronization steps 26
8.2. Display Copy Mode 27
12.1. The 3 font runtime style transformations (filters). 46

1 Bibliography

[CM3REF]	STM32JavaF2 ARMCCv4 Reference Manual (TLT-0611-REF-STM32JavaF2)
[CM3MAN]	STM32JavaF2 ARMCCv4 User's Manual (TLT-0612-MAN-STM32JavaF2)
[CM3UIREF]	STM32JavaF2 ARMCCv4 User Interface Extension Reference Manual (TLT-0613-REF-STM32JavaF2)
[B-ON]	Beyond: ESR 001, 2008, http://www.e-s-r.net
[MUI]	Micro User Interface: ESR 002, 2009, http://www.e-s-r.net
[MWT]	Micro Widgets Toolkit: ESR 011, 2010, http://www.e-s-r.net

2 Introduction

2.1 Scope

This document explains how the user interface features of STM32JavaF2 ARMCCv4 are accessed, configured and used in the MicroEJ workbench. It describes how to extend a Java platform to support user interfaces, including an outline of the required low-level drivers. It also describes how to create a Java application that has a user interface, and test it with the simulator.

2.2 Intended audience

The audience for this document is software engineers who need to understand the details of the JPF User Interface Extension components, including their APIs, error codes and options.

2.3 Related Documents

Please refer to [CM3UIREF] for details of the User Interface Extension components, and to [CM3MAN] for information about creating and configuring a JPF.

2.4 Document Organization

The document is divided into several parts:

- The design principles of platforms that support user interfaces
- How the MicroUI library is configured and extended
- The modules available to add user interface capabilities to a Java platform
- How to configure the simulator to support user interface features
- How to use the customize a Java application launcher.

3 Platform Design

The STM32JavaF2 ARMCCv4 User Interface Extension features one of the fastest graphical engines, associated with a unique int based event management system. It provides [MUI] and [MWT] library implementations. The following diagram depicts the components involved in its design, along with the provided tools:

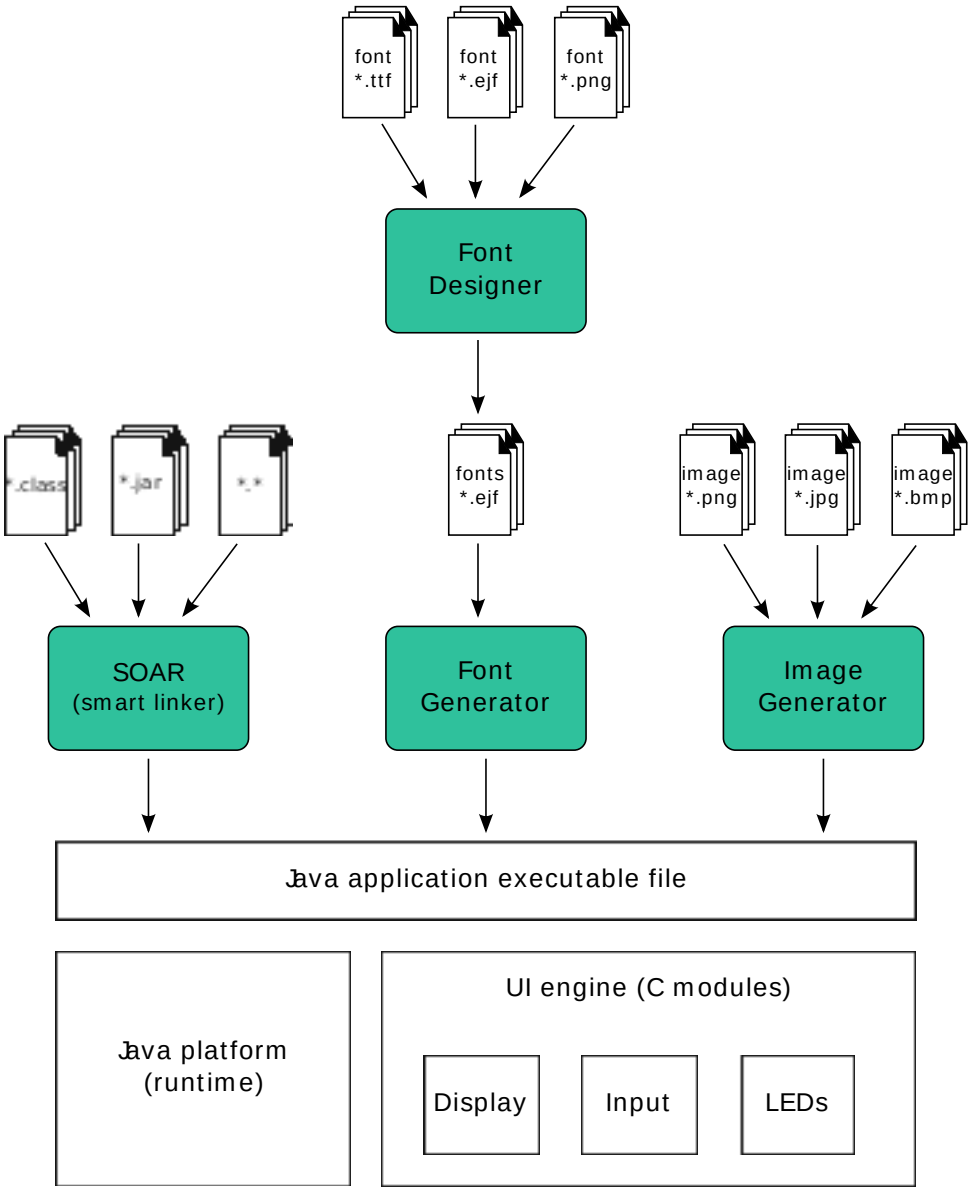


Figure 3.1. The user interface extension components along with a STM32JavaF2 ARMCCv4 platform

The diagram below shows a simplified view of the components involved in the provision of a Java user interface, in both the embedded (EmbJPF) and simulated (SimJPF) modes.

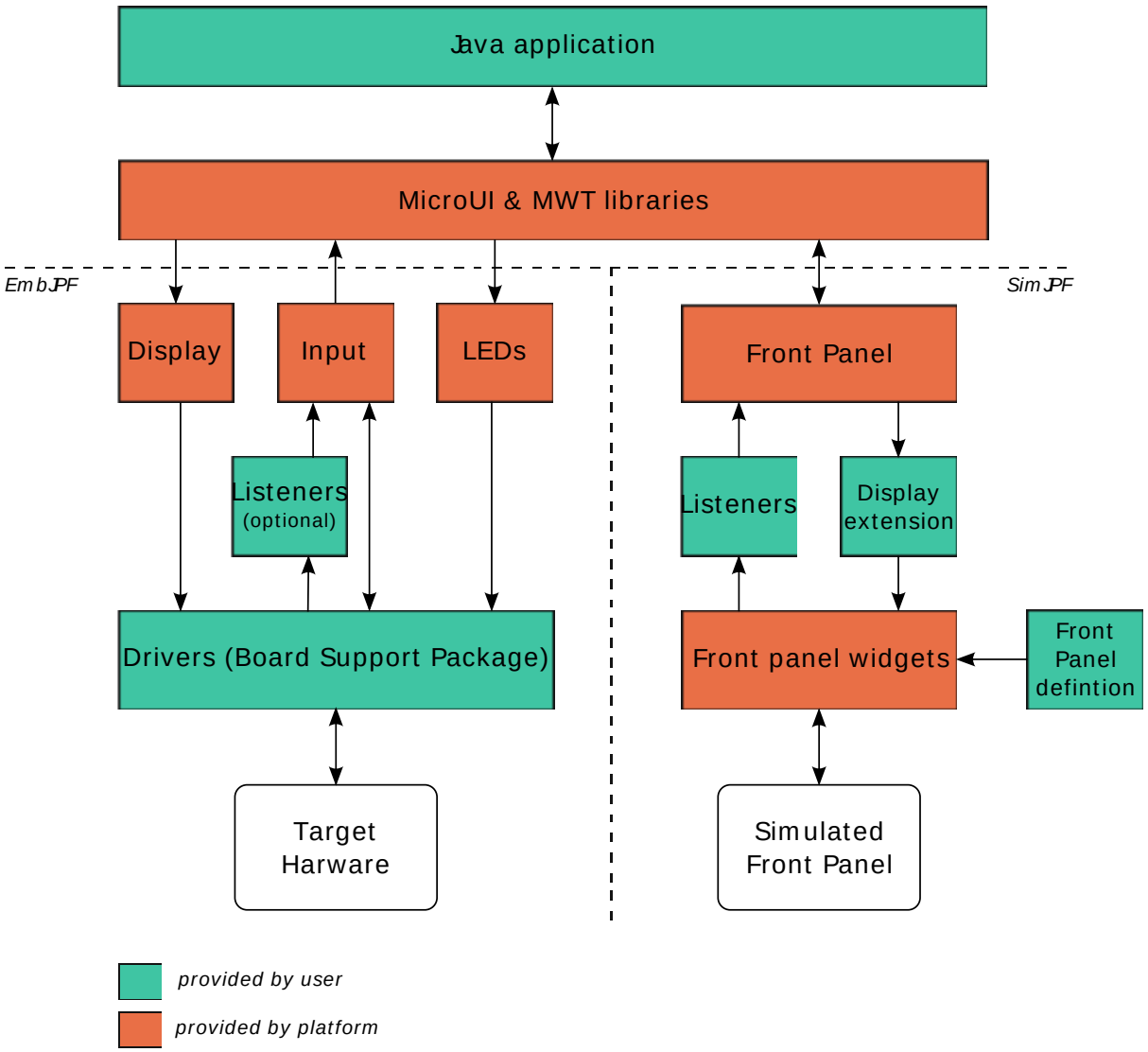


Figure 3.2. Overview

Stacks are the native parts of MicroUI. They connect the Java MicroUI library to the user-supplied drivers code (coded in C).

Drivers for input devices must generate events that are sent, via a MicroUI *Event Generator*, to the Java application. An event generator accepts notifications from devices and generates an event in a standard format that can be handled by the application. Depending on the MicroUI configuration, there can be several different types of event generator in the system, and one or more instances of each type. Each instance has an unique id.

Drivers may either interface directly with event generators, or they can send their notifications to a *Listener*, also written in C, and the listener passes the notifications to the event generator. This decoupling has two major benefits:

- The drivers are isolated from the MicroEJ libraries – they can even be existing code.
- The listener can translate the notification, so, for example, a joystick could generate pointer events.

For the SimJPF, the platform is supplied with a set of software widgets that generically support a range of input devices, such as buttons, joysticks and touchscreens, and output devices such as pixelated displays and LEDs. With the help of the Front Panel Designer tool that forms part of the MicroEJ workbench the user must define a front panel mock-up using these widgets. The user must provide a set of listeners

that connect the input widgets to event generators. The user may choose to simulate events that will ultimately come from a special-purpose input device using one of the standard input widgets; the listener will do the necessary translation. The user must also supply, in Java, a display extension that adapts the supplied display widget to the specifics of the hardware being simulated.

4 MicroUI

4.1 Introduction

The MicroUI module defines a low-level UI framework for embedded devices. This module allows the creation of basic Human-Machine-Interfaces (HMI), with output on a pixelated screen. For more information please consult the MUI Specification [MUI].

4.2 Architecture

MicroUI is not a standalone library. It requires a configuration step and several extensions to drive I/O devices (display, inputs, LEDs etc.).

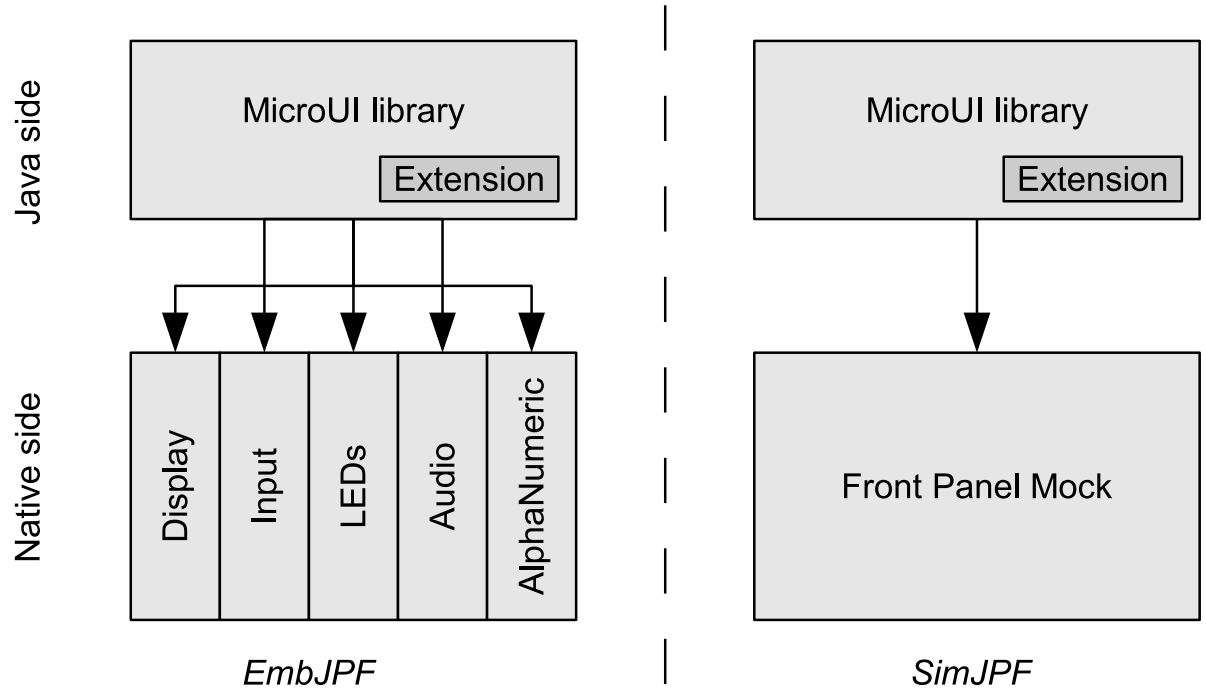


Figure 4.1. MicroUI Elements

At Java application startup all MicroUI objects relative to the I/O devices are created and accessible. The following MicroUI methods allow to access to these internal objects:

- `Display.getDefaultDisplay()`: returns the instance of the default display which drives the main LCD screen.
- `Leds.getNumberOfLeds()`: returns the numbers of available LEDs.

First of all MicroUI requires a configuration step to create these internal objects before the call to the `main()` method. The chapter “Static Initialization” explains how to perform the configuration step. Note that this configuration step is the same for the embedded and simulation platforms.

The embedded platform requires some additional C libraries to drive the I/O devices. Each C library is dedicated to a specific kind of I/O device. A specific chapter is available for each kind of I/O devices.

I/O devices	Extension Name	Chapter
Graphical / pixelated display (LCD screen)	Display	Section 8
Inputs (buttons, joystick, touch, pointers etc.)	Input	Section 7
LEDs	LEDs	Section 6
Audio tone	Audio	Section 9

I/O devices	Extension Name	Chapter
Alphanumeric display (character display)	AlphaNumeric	Section 10

Table 4.1. MicroUI C libraries

The simulation platform uses a mock which simulates all I/O devices. Refer to the chapter “Simulation”.

4.3 MicroUI Internal

The MicroUI module provides the MicroUI library API (see MUI Specification [MUI]) and an additional API called "MicroUI-Internal" API.

The MicroUI Internal library has the same function as the MicroUI library but it exposes additional APIs (MicroUI Internal API is a super set of standard MicroUI API) to support extending the MicroUI library. These APIs allow to the user to create his/her own events generators and connect them to the C world.

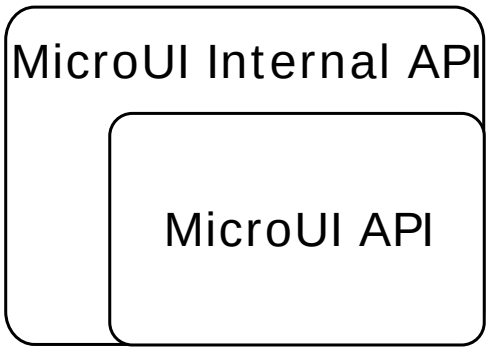


Figure 4.2. MicroUI APIs

To define a special-purpose event generator the Java application project has to contain a class which extends the abstract class `com.is2t.microui.io.EventGenerator` whose name matches that specified in the `eventgenerator` tag of the MicroUI static initialization XML file (refer to chapter (see “Static Initialization”).

```
<eventgenerator name="SPECIAL_PURPOSE"
   class="foo.bar.Zork">
   <property name="PROP1" value="3"/>
   <property name="PROP2" value="aaa"/>
   </eventgenerator>
```

Figure 4.3. User-defined Event Generator Declaration

This event generator instance will be instantiated during the MicroUI initialization step (see “Static Initialization”). At runtime it will receive all dedicated events sent by the C side. If the Java application does not contain this extension class, the event generator will be unusable and a warning will be thrown during the MicroUI initialization step.

Notes:

- The abstract class `com.is2t.microui.io.EventGenerator` is not available in the standard MicroUI API but its implementation is available in both the MicroUI library and in the MicroUI-Internal library.
- A classpath variable named `MICROUI_INTERNAL-1.4.1` is available; add it to your Java build path project to target the MicroUI internal library. Add this classpath variable into the Java application classpath in order to be able to extend the abstract class `com.is2t.microui.io.EventGenerator`.

4.4 Threads

MicroUI implementation for MicroEJ uses internal threads. These threads are created during the MicroUI initialization step and started automatically before the call to the Java application `main()` method

or not, according the option `autostart` (see MicroUI options in Section 18). Refer the the MicroUI specification [MUI] to have more information about the internal threads.

- DisplayPump: this thread manages all display events (`repaint`, `show()` etc. There is a thread per display.

  InputPump: this thread reads the I/O devices inputs and dispatches them into the display pump(s).

The threads are always running. The user has to account with them to determine the number of concurrent threads the MicroJvm Virtual Machine can run (see Memory options in Section 18).

The threads cannot be stopped with a Java exception. The exceptions are always checked and printed as an error message via `System.out`.

4.5 Layers

4.5.1 Principle

`MicroUI-Layers` Java library is an additional library of `MicroUI`. This library provides some APIs to stack several images [1] on a working buffer [2] different than the LCD working buffer before flushing all working buffer data into the LCD buffer.

The main idea is to manipulate some images (we talk about "layers") between themselves without drawing into. We can do the parallel with an images software for computers such as GIMP.

4.5.2 API

First of all, Java application has to contain an implementation of the abstract class `ej.microui.layer.AbstractLayersManager`. This implementation is dedicated to a `Display` and it is used to define the order of the layers.

To enable a layers manager the application has to call the method `show` of the layers manager. The MicroUI framework will call the layers manager `stackLayers` method just after a `repaint` and just before every flush. This allows to the application to update its rendering as usual before stacking its layers.

Before stacking the very first layer the application should fix the working buffer background color calling `setBackgroundColor` method. Then it can stack the first layer fixing:

- its position: the layer can be stacked anywhere on the working buffer,
- its size: the full layer area or only a part of the layer can be stacked,
- its transparency: the layer is merged with the working buffer respecting a color transformation according a transparency level (see “Transparency”).

Note that a same layer can be stacked several times with or without the same parameters.

When all layers have been stacked, MicroUI framework replaces the LCD working buffer content by the layers working buffer. The same operation has to be done again at every call to `stackLayers`. The application can stop the use of the layers manager calling the `hide` method.

Some additional APIs are available:

- `getLayersManager`: A `Display` can hold only one layers manager at any time. This method returns the current layers manager.

[1] MicroUI `Image` created from a resource (`PNG` images etc.), MicroUI `Image` manually created at runtime (with a custom width and height) and MicroUI `Display`.
[2] The working buffer is a buffer whose size is the same size than the buffer used to store the LCD data.

- `setTransparency` and `getTransparency`: see “Transparency”.
- `createImage`: these APIs allow to make a screenshot of the actual content of a `Display`. These new images can be stacked as layers later.
- `flush`: the application can ask to the MicroUI framework to produce a "flush" (and so a call to `stackLayers`).

4.5.3 Transparency

There are two ways to use the transparency on a layer. The both ways are complementary: stack a layer fixing a global transparency level and draw with transparency.

4.5.3.1 Layer Transparency Level

When a layer is stacked, a transparency level can be set. This a value between 0 and 255 when 0 means "fully transparent" and 255 "fully opaque".

When null or 0, the layer is not stacked (useless). Otherwise a merge between each layer pixel (called foreground pixel) and each working buffer targeted pixel (called background pixel) is done. The merge follows this formula:

$$\alpha Mult = (\alpha FG * \alpha BG) / 255$$
$$\alpha Out = \alpha FG + \alpha BG - \alpha Mult$$
$$COut = (CFG * \alpha FG + CBG * \alpha BG - CBG * \alpha Mult) / \alpha Out$$

where:

- αFG is the alpha level of the foreground pixel (layer pixel)
- αBG is the alpha level of the background pixel (working buffer pixel)
- C*xx* is a color component of a pixel (Red, Green or Blue).
- αOut is the alpha level of the final pixel

4.5.3.2 Draw with Transparency

As mentioned in previous chapter it is possible to manage the transparent pixels available in an image. Note that this feature is only available when the native pixel format (LCD buffer pixel format) can hold the transparency level (only the mode 32 bits per pixel).

A MicroUI `Image` created from a resource (PNG image etc.) can already contain some transparent pixels. So when this image is used as layer, its pixels will be merged respecting the previous formula.

For the other images (created at runtime using a free width and height) and for the display area, it is possible to fix the alpha rendering for the next drawings. The API `AbstractLayersManager.setTransparency` replaces the default alpha level (full opaque: 255). When the image (or the display) is used as a layer, the previous formula is respected. If the image (or the display) is used as usual (not as a layer), this alpha level is not taken in consideration.

4.5.4 Implementation

The layers management requires a specific MicroUI framework implementation. This implementation is done is the display module (see “Display”).

The standard display modules (`switch`, `copy`, `direct` and `custom`) implement the layers extension in software. The display implementation `hwlayers` does not implement the layer extension and provides some additional low level APIs to let the LCD driver manage the layers rendering.

This additional low level APIs allow to use the LCD and/or LCD controller features such as some rendering accelerator.

4.5.5 Dependencies

- MicroUI module.

4.5.6 Installation

MicroUI Layers API is automatically installed during the MicroUI module installation.

4.5.7 Use

The classpath variable named `MICROUI-LAYERS-1.0.0`

4.6 Transparency

MicroUI provides several policies to use the transparency. These policies depends on several factors: the kind of drawing, the LCD pixel rendering format, the layers mode. The main concept is that MicroUI does not allow to draw something with a transparency level different than 255 (fully opaque). There are three exceptions: the images, the fonts and the layers mode.

4.6.1 Images

Drawing an image (a pre-generated image or an image decoded at runtime) which contains some transparency level depends on the LCD pixel rendering format. After the decoding step the image stored in memory respects the same pixel rendering format than the LCD.

4.6.1.1 32 bits-per-pixel

This pixel format contains 8 bits to store the transparency level (alpha). This byte is used to merge the foreground pixel (image transparent pixel) with the background pixel (LCD buffer opaque pixel). The formula to obtain the pixel is:

```
αMult = (αFG * αBG) / 255
αOut = αFG + αBG - αMult
COut = (CFG * αFG + CBG * αBG - CBG * αMult) / αOut
```

where:

- α*FG* is the alpha level of the foreground pixel (layer pixel)
- α*BG* is the alpha level of the background pixel (working buffer pixel)
- C*xx* is a color component of a pixel (Red, Green or Blue).
- α*Out* is the alpha level of the final pixel

4.6.1.2 Other Pixel Formats

All others formats (16, 8, 4, 2, 1 bits-per-pixel) do not store in the pixel word a peace of information about the transparency. MicroUI module keeps an extra bit to store the transparency level for each pixel. This extra bit is an internal MicroUI module and it is not accessible by the low level driver.

This extra bit allows to know if the pixel is fully opaque ('1' for 255) or fully transparent ('0' for 0). It so possible to draw images which contain this kind of information.

4.6.2 Fonts

Contrary to the images, the fonts can be rendered with a fixed alpha transparency level without any link with the LCD pixel rendering format. This fixed alpha level is defined during the pre-generation of a font (see “Fonts”).

- '1' means 2 levels are managed: fully opaque and fully transparent.
- '2' means 4 levels are managed: fully opaque, fully transparent and 2 intermediate levels.
- '4' means 16 levels are managed: fully opaque, fully transparent and 14 intermediate levels.
- '8' means 256 levels are managed: fully opaque, fully transparent and 254 intermediate levels.

4.6.3 Layers

The layers mode allows to use the transparency to draw something (a line, etc.). Please refer to the chapter “Layers” to have more information.

4.7 Dependencies

- EDC Java core module (see [TLT-0612-MAN-STM32JavaF2]).
- B-ON Java core module (see [TLT-0612-MAN-STM32JavaF2]).
- MicroUI initialization step (see “Static Initialization”).
- MicroUI C libraries (see “Architecture”).

4.8 Installation

MicroUI Java library is an additional module. In the platform configuration file, check `UI > MicroUI` to install it. When checked, the XML file `microui > microui.xml` is required during platform creation to configure the module. This configuration step is used to extend the MicroUI library. Refer to the chapter “Static Initialization” to have more information about the MicroUI Initialization step.

4.9 Use

Two classpath variables named `MICROUI-1.4.1` and `MICROUI-INTERNAL-1.4.1` (see “MicroUI Internal”) are available.

This library provides a set of options. Refer to the chapter “Launch Options” which lists all options.

5 Static Initialization

5.1 Introduction

MicroUI requires a configuration step (also called extension step) to customize itself before Java application startup (see “Architecture”). This configuration step uses an XML file. In order to save both runtime execution time and flash memory, the file is processed by the Static MicroUI Initializer tool, avoiding the need to process the XML configuration file at runtime. The tool generates appropriate initialized objects directly within the MicroUI library, as well as Java and C constants files for sharing MicroUI event generators IDs.

This XML file (also called initialization file) defines:

- the MicroUI event generators that will exist in the application, in relation to low level drivers that provide data to these event generators (see “Inputs”) .
- whether the application has a display, and if so, it provides its logical name.
- which fonts will be provided to the application.

5.2 Functional Description

The Static MicroUI Initializer tool takes as entry point the initialization file which describes the MicroUI library extension. This tool is automatically launched during the MicroUI module installation (see “Installation”).

The Static MicroUI Initializer tool is able to out until three files:

- A Java library which extends MicroUI library. This library is automatically added to the Java application classpath when MicroUI library is set as a classpath variable. This library is used at MicroUI startup to create all instances of I/O devices (`Display`, `EventGenerator` etc.) and contains the fonts described into the configuration file (these fonts are also called "system fonts").

  This MicroUI extension library is always generated and MicroUI library cannot run without this extension.

- A C header file (*.h) file. This H file contains some IDs which are used to make a link between an input device (buttons, touch) and its MicroUI event generator (see “Inputs”).

  This file is useless if the BSP does not provide any input device and the Static MicroUI Initializer tool is able to not generate this file. Otherwise the MicroUI configuration file has to specify where put this file, typically in a BSP `include` directory.

- A Java interface file file. This Java file contains the same IDs which are used to make a link between an input device (buttons, touch) and its MicroUI event generator (see “Inputs”).

  This Java file is used to configure the simulator with the same characteristics as the BSP.

  This file is useless if the BSP does not provide any input device and the Static MicroUI Initializer tool is able to not generate this file. Otherwise the MicroUI configuration file has to specify where put this file, typically in the simulator project (also called front panel project, see “Simulation”).

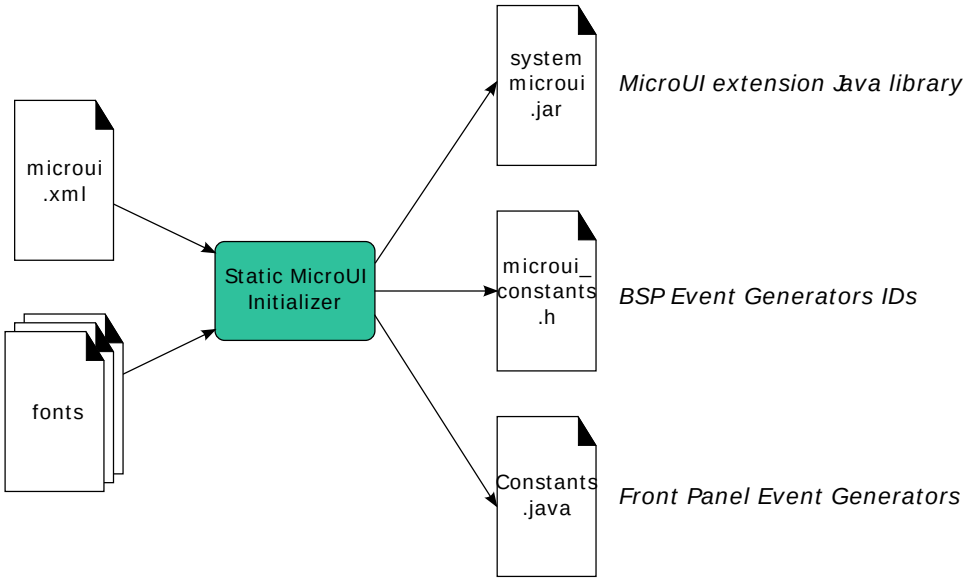


Figure 5.1. MicroUI Process

5.3 Root Element

The initialization file root element is `<microui>` and contains component specific elements.

```
<microui>
    [ component specific elements ]
</microui>
```

Figure 5.2. Root Element

5.4 Display Element

The display component augments the initialization file with:

- the configuration of each display
- fonts that are implicitly embedded within the application (also called system fonts). Applications can also embed their own fonts.

```
<display name="DISPLAY"/>

<fonts>
    <font file="resources\fonts\myfont.ejf">
        <range name="LATIN" sections="0-2"/>
        <customrange start="0x21" end="0x3f"/>
    </font>
    <font file="C:\data\myfont.ejf"/>
</fonts>
```

Figure 5.3. Display Element

5.5 Event Generators Element

The event generators component augments the initialization file with:

- the configuration of predefined MicroUI `Event Generator`: `Command`, `Buttons`, `States`, `Pointer`, `Touch`
- the configuration of generic MicroUI `Event Generator` (see “MicroUI Internal”)

```
<eventgenerators>
 <!-- Generic Event Generators -->
    <eventgenerator name="GENERIC" class="foo.bar.Zork">
        <property name="PROP1" value="3"/>
        <property name="PROP2" value="aaa"/>
    </eventgenerator>

    <!-- Predefined Event Generators -->
    <command name="COMMANDS"/>
    <buttons name="BUTTONS" extended="3"/>
    <buttons name="JOYSTICK" extended="5"/>
    <pointer name="POINTER" width="1200" height="1200"/>
    <touch name="TOUCH" display="DISPLAY"/>
    <states name="STATES" numbers="NUMBERS" values="VALUES"/>

</eventgenerators>

<array name="NUMBERS">
    <elem value="3"/>
    <elem value="2"/>
    <elem value="5"/>
</array>

<array name="VALUES">
    <elem value="2"/>
    <elem value="0"/>
    <elem value="1"/>
</array>
```

Figure 5.4. Event Generator Element

5.6 Example

This common MicroUI initialization file initializes MicroUI with:

- a display,
- a `Command` event generator,
- a `Buttons` event generator which targets *n* buttons (3 first buttons having extended features),
- a `Buttons` event generator which targets the buttons of a joystick,
- a `Pointer` event generator which targets a touch panel,
- a `DisplayFont` whose path is relative to this file.

```
<microui>

 <display name="DISPLAY"/>

 <eventgenerators>
     <command name="COMMANDS"/>
     <buttons name="BUTTONS" extended="3"/>
     <buttons name="JOYSTICK" extended="5"/>
     <touch name="TOUCH" display="DISPLAY"/>
 </eventgenerators>

 <fonts>
     <font file="resources\fonts\myfont.ejf"/>
 </fonts>

</microui>
```

Figure 5.5. MicroUI Initialization File Example

5.7 Dependencies

No dependency is required.

5.8 Installation

The Static Initialization tool is part of MicroUI module (see “MicroUI”). Install the MicroUI module to install the Static Initialization tool and fill all properties in MicroUI module configuration file (which has to specify the name of the initialization file).

5.9 Use

The Static MicroUI Initializer tool is automatically launched during the MicroUI module installation.

6 LEDs

6.1 Introduction

The LEDs module contains the C part of the MicroUI implementation which manages LED devices. This module is composed of two elements:

- the C part of MicroUI LEDs API (a built-in C archive),
- an implementation of low level API for the LEDs (LLLEDS) the BSP must provide.

6.2 Implementations

LEDs module provides only one implementations which exposes some low level API (LLEDS) to let the BSP manages the LEDs. This implementation of the MicroUI `Leds` API provides some low level API. The BSP has to implement these LLAPI making the link between the MicroUI C library `leds` and the BSP LEDs drivers.

The LLAPI to implement are listed in the header file `LLLEDS_impl.h`. First of all, in the initialization function, the BSP has to return the available number of LEDs the board provides. The others functions are used to turn on and off the LEDs.

The LLAPI are the same for the LED which is connected to a `GPIO` (`0` or `1`) or via a `PWM`. The BSP has the responsability to interpret the Java application parameter `intensity`.

Typically, when the LED is connected to a `GPIO`, the `intensity` "0" means "OFF" and all others values "ON". When the LED is connected via a `PWM`, the `intensity` "0" means "OFF" and all others values must configure the `PWM` signal.

The BSP should be able to return the state of a LED. If it is not able to do it (for instance `GPIO` is not accessible in read mode), the returned value may be wrong. The Java application may not be able to know the LEDs states.

When there is no LED on the board, a *stub* implementation of C library is available. This C library must be linked by the third-party C IDE when MicroUI module is installed in the Java platform.

6.3 Dependencies

- MicroUI module (see “MicroUI”)
- `LLLEDS_impl.h` implementation if standard implementation is chosen (see “Implementations”).

6.4 Installation

LEDs is a sub-part of MicroUI Java library. When MicroUI module is installed, the LEDs module should be installed too, otherwise it may throw linker errors during Java application compilation.

In the platform configuration file, check `UI` > `LEDs` to install it.

6.5 Use

The MicroUI LEDs APIs are available in the class `ej.microui.io.Leds`.

7 Inputs

7.1 Introduction

The Inputs module contains the C part of the MicroUI implementation which manages input devices. This module is composed of two elements:

- the C part of MicroUI input API (a built-in C archive),
- an implementation of low level API for the input devices (LLINPUT) the BSP must provide.

7.2 Functional Description

The Inputs module implements the MicroUI `int`-based event generators framework. `LLINPUT` specifies the low level API to send events to the Java world.

Each MicroUI `Event Generator` represents one side of a pair of collaborative components that communicate using a shared buffer:

- the producer: the C driver connected to the hardware. As a producer, it sends its data into the communication buffer.
- the consumer: the MicroUI `Event Generator`. As a consumer, it reads (and removes) the data from the communication buffer.

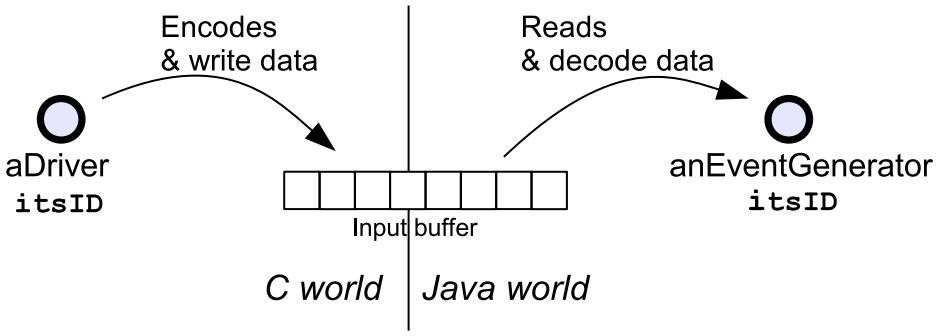


Figure 7.1. Drivers and MicroUI Event Generators Communication

The `LLINPUT` API allows multiple pairs of `<driver - event generator>` to use the same buffer, and associates drivers and event generators using an int ID. The ID used is the event generator ID held within the MicroUI global registry [MUI]. Apart from sharing the ID used to "connect" one driver data to its respective event generator, both entities are completely decoupled.

A Java green thread, called the `InputPump` thread, waits for data to be published by drivers into the "input buffer", and dispatches to the correct (according to the ID) event generator to read the received data. This "driver-specific-data" is then transformed into MicroUI events by event generators and sent to objects that listen for input activity.

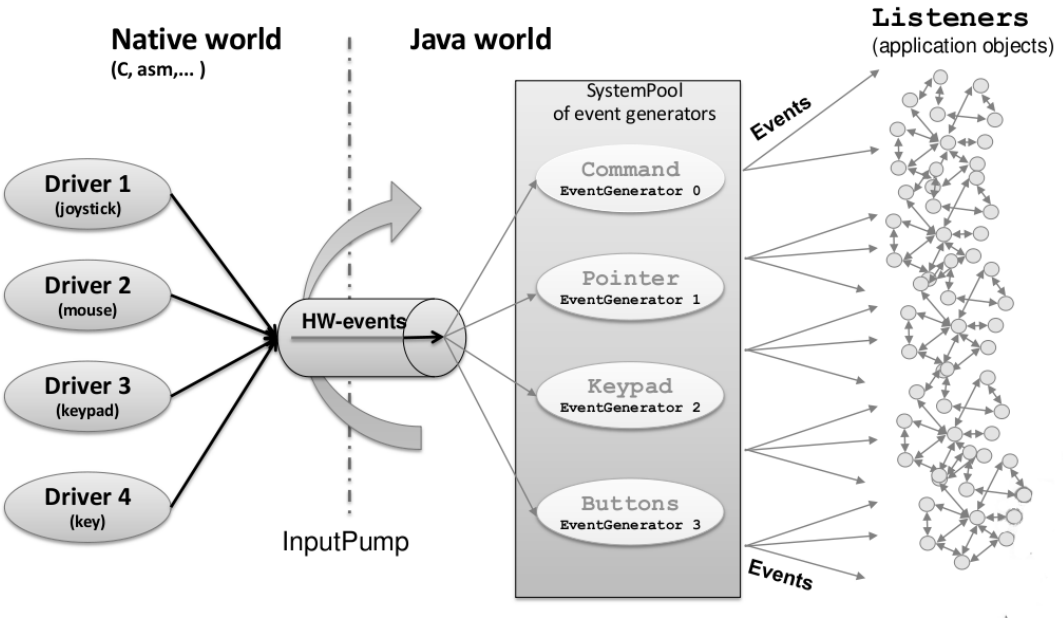


Figure 7.2. MicroUI Events Framework

7.3 Implementations

Inputs module provides two implementations. A *default* one which exposes some low level API (LLINPUT) to let the BSP manages the input devices and an *advanced* one which adds the notion of listeners between the input devices drivers.

When there is no input device on the board, a *stub* implementation of C library is available. This C library must be linked by the third-party C IDE when MicroUI module is installed in the Java platform.

7.3.1 Default

This implementation of the MicroUI `Event Generator` APIs provides some low level API. The BSP has to implement these LLAPI making the link between the MicroUI C library `inputs` and the BSP input devices drivers.

The LLAPI to implement are listed in the header file `LLINPUT_impl.h`. It allows events to be sent to the MicroUI implementation. The input drivers are allowed to add events directly using the event generator's unique ID (see “Static Initialization”). The drivers are fully dependent on the MicroEJ framework (a driver cannot be developed without MicroEJ because it uses the header file generated during the MicroUI initialization step).

7.3.2 Advanced

This implementation of the MicroUI `Event Generator` APIs provides the same low level API than the standard implementation. The BSP has to implement these LLAPI making the link between the MicroUI C library `inputs` and the BSP input devices drivers.

The LLAPI to implement are listed in the header file `LLINPUT_impl.h`. It allows events to be sent to the MicroUI implementation. This implementation supports the use of listeners between the input drivers and the MicroUI `Event Generator`. This allows the input drivers to be developed independently of the MicroEJ framework. The listener implementations have to convert the driver’s native events into MicroUI events.

Furthermore, this implementation allows to simulate easily an input device with another. For instance it is possible to simulate a `Pointer` with a joystick. Instead of developing the transformation of joystick events in pointer events directly in the joystick driver (which makes more complex the write of joystick driver), the transformation can be done in the joystick listener file.

7.4 Dependencies

- MicroUI module (see “MicroUI”)
- Static MicroUI initialization step (see “Static Initialization”). This step generates an header file which contains some unique event generator IDs. These IDs must be used in the BSP to make the link between the input devices drivers and the MicroUI `Event Generators`.
- `LLINPUT_impl.h` implementation if standard or advanced implementation is chosen (see “Implementations”).

7.5 Installation

Inputs is a sub-part of MicroUI Java library. When MicroUI module is installed, the Inputs module should be installed too, otherwise it may throw linker errors during Java application compilation.

In the platform configuration file, check `UI > Inputs` to install it. When checked, the properties file `inputs > inputs .properties` is required during platform creation to configure the module. This configuration step is used to choose the kind of implementation (see “Implementations”).

The properties file have to / can contain the following properties:

- `type` [optional, default value is "default"]: Defines the type of Inputs implementation. Two types are available. All others types are forbidden (throw a generation error).
  - `default`: select this configuration to use the standard Inputs implementation.
  - `advanced`: select this configuration to use the Inputs implementation which allows to separate the input devices drivers and the input listeners.

7.6 Use

The MicroUI Input APIs are available in the class `ej.microui.EventGenerator`.

8 Display

8.1 Introduction

The Display module contains the C part of the MicroUI implementation which manages graphical displays. This module is composed of two elements:

- the C part of MicroUI Display API (a built-in C archive),
- an implementation of low level API for the displays (LLDISPLAY) the BSP must provide.

MicroUI is multi-display oriented, as is the API of the Display module for the low level `LLDISPLAY` API.

8.2 Display Configurations

The Display modules provides a number of different configurations. The appropriate configuration should be selected depending on the capabilities of the screen and other related hardware such as LCD controllers.

The modes can vary in three ways:

- the buffer mode: double-buffer, simple buffer (also known as "direct"), custom
- the memory layout of the pixels
- pixel depth

The supplied configurations offer a limited range of combinations of the options.

8.3 Buffer Modes

When using the double buffering technique, the memory into which the application draws (called the back buffer) is not the memory used by the screen to refresh it (the display buffer). When everything has been drawn consistently from the application point of view, the back buffer content is synchronized with the display buffer. Double buffering avoids flickering and inconsistent rendering: it is well suited to high quality animations.

For more static display-based applications, and/or to save memory, an alternative configuration is to use only one buffer, shared by both the application and the screen.

Displays addressed by one of the standard configuration are called *generic displays*. For these generic displays, there are three buffer modes: switch, copy, and direct.

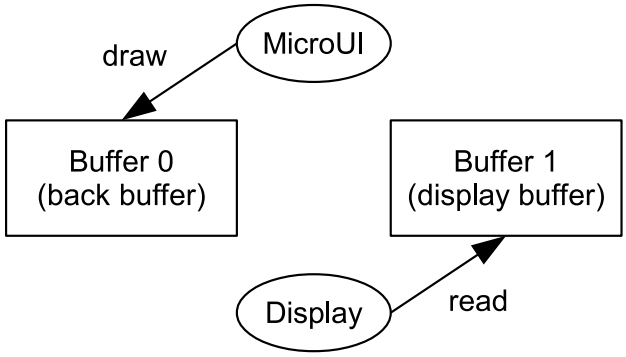
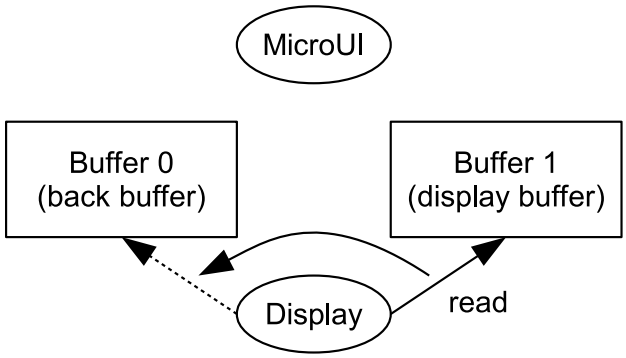
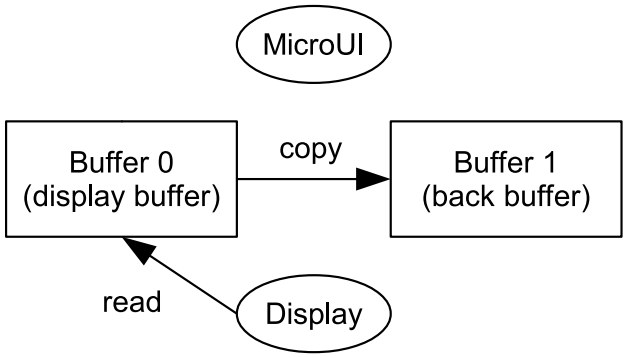
For cases where the display memory cannot be accessed directly, or where the memory layout is non-standard, a custom configuration provides the means to connect any kind of "exotic" display.

8.3.1 Switch

The switch mode is a double-buffered mode where two buffers in RAM alternately play the role of the back buffer and the display buffer. The display source is alternatively changed from one buffer to the other.

Switching the source address may be done asynchronously. The synchronize function is called before starting the next set of draw operations and must wait until the driver has switched to the new buffer.

Synchronization steps are described in Table 8.1.

Step 1: Drawing MicroUI is drawing in buffer 0 (back buffer) and the display is reading its content from buffer 1 (display buffer).	
Step 2: Switch The drawing is done. Set that the next read will be done from buffer 0. Note that the display "hardware component" asynchronously continues to read data from buffer 1.	
Step 3: Copy A copy from the buffer 0 (new display buffer) to the buffer 1 (new back buffer) must be done to keep the content of current drawing. The copy routine must wait until the display has finished the switch and start asynchronously by comparison with the MicroUI drawing routine (see next step). This copy routine can be done in a dedicated RTOS task or in an interrupt routine. The copy should start after the display "hardware component" has finished a full buffer read to avoid flickering. Usually a tearing signal from the LCD at the end of the read of the previous buffer (buffer 1) or at the beginning of the read of the new buffer (buffer 0) throws an interrupt. The interrupt routine starts the copy using a DMA. If it is not possible to start an asynchronous copy, the copy must be performed in the MicroUI drawing routine, at the beginning of next step. Note that the copy is partial: only the parts that have changed need to be copied, lowering the CPU load.	
Step 4: Synchronization	

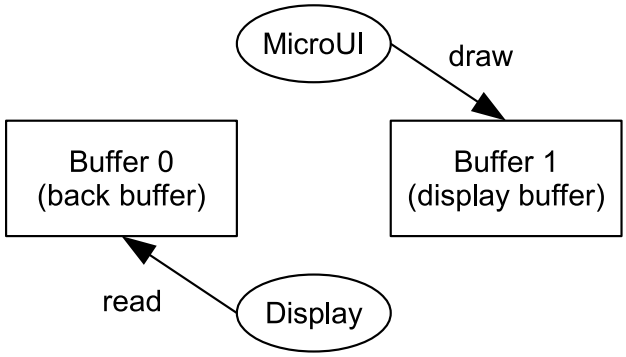
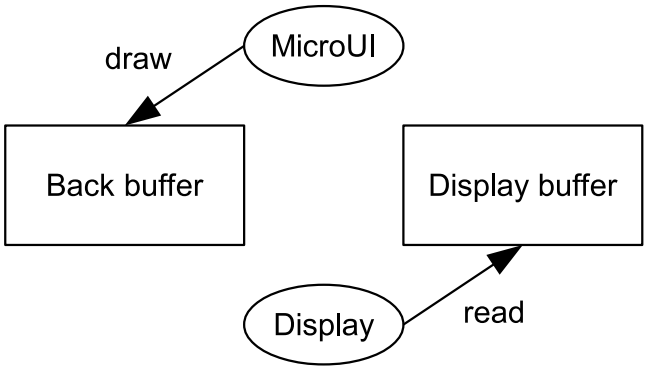
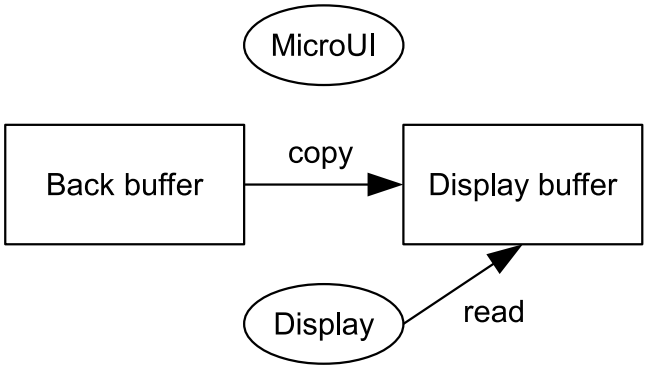
Waits until the copy routine has finished the full copy. If the copy has not been done asynchronously, the copy must start after the display has finished the switch. It is a blocking copy because the next drawing operation has to wait until this copy is done.	
Step 4: Next draw operation Same behavior as step 1 with buffers reversed.	

Table 8.1. Switch mode synchronization steps

8.3.2 Copy

The copy mode is a double-buffered mode where the back buffer is in RAM and has a fixed address. To update the display, data is sent to the display buffer. This can be done either by a memory copy or by sending bytes using a bus such as SPI or I2C.

Synchronization steps are described in Table 8.2.

Step 1: Drawing MicroUI is drawing in the back buffer and the display is reading its content from the display buffer.	
Step 2: Copy The drawing is done. A copy from the back buffer to the display buffer is triggered. Note that the implementation of the copy operation may be done asynchronously – it is recommended to wait until the display "hardware component" has finished a full buffer read to avoid flickering. At the implementation level, the copy may be done by a DMA, a dedicated RTOS task, interrupt etc.	

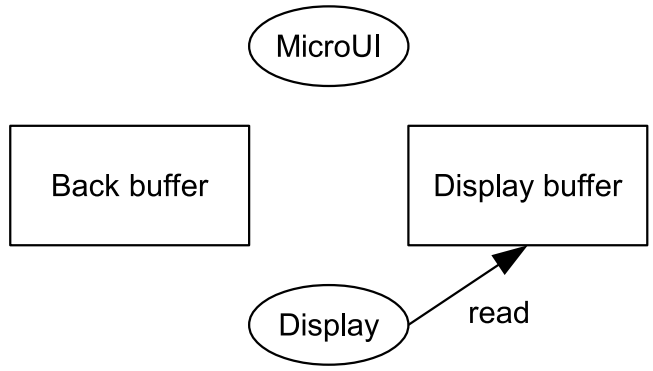
Step 3: Synchronization The next drawing operation waits until the copy is complete.	

Table 8.2. Display Copy Mode

8.3.3 Direct

The direct mode is a single-buffered mode where the same memory area is used for the back buffer and the display buffer (Figure 8.1). Use of the direct mode is likely to result in "noisy" rendering and flickering, but saves one buffer in runtime memory.

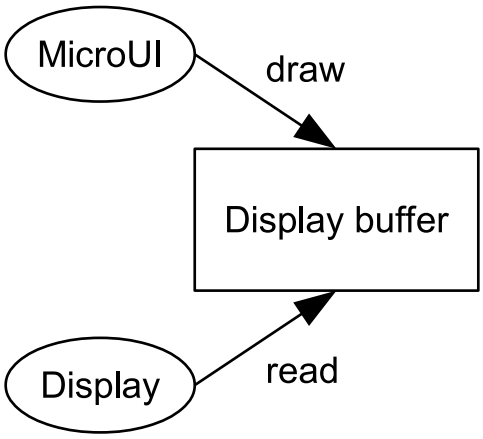


Figure 8.1. Display Direct Mode

8.3.4 Custom

When the display buffer is not accessible from memory, a custom display configuration provides access to primitive functions such as drawing/reading a pixel, or a rectangle. This mode should be avoided as it results in slow pixel-by-pixel operations.

8.4 Memory Layout

The Display module provides generic display configurations to support one of two memory organizations:

- Line by line (also called XY - X before Y): pixels are laid out from left to right within a line, starting with the top line. For a display with 16 bits-per-pixel, the pixel at (0,0) is stored at memory address 0, the pixel at (1,0) is stored at address 2, the pixel at (2,0) is stored at address 4, and so on.

- Column by column (also called YX - Y before X): pixels are laid out from top to bottom within a column, starting with the left column. For a display with 16 bits-per-pixel, the pixel at (0,0) is stored at memory address 0, the pixel at (0,1) is stored at address 2, the pixel at (0,2) is stored at address 4,and so on.

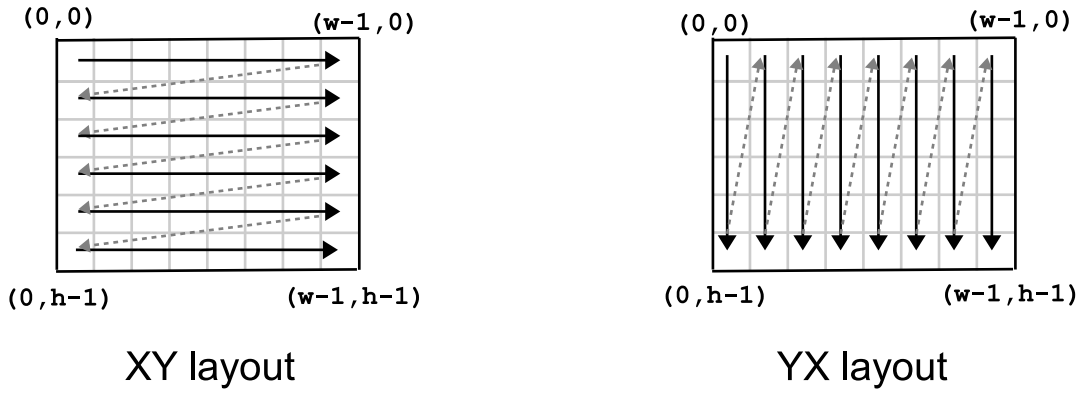


Figure 8.2. Memory layouts

8.5 Pixel Structure

The Display module provides pre-built display configurations with generic pixel memory layout. The layout of the bits within the pixel is driver specific. The driver must implement functions that convert between MicroUI's standard 24 bit RGB layout and the display's own pixel layout.

8.6 Implementations

Display module provides two implementations. A *default* one which exposes some low level API (LLDISPLAY) to let the BSP manages the display and a *custom* one dedicated to non-generic display which exposes additional APIs to perform some drawings.

When there is no display on the board, a *stub* implementation of C library is available. This C library must be linked by the third-party C IDE when MicroUI module is installed in the Java platform.

8.6.1 Default

This implementation of the MicroUI `Display` API targets generic display (see “Display Configurations”): Switch, Copy and Direct. It provides some low level API. The BSP has to implement these LLAPI making the link between the MicroUI C library `display` and the BSP display driver.

The LLAPI to implement are listed in the header file `LLDISPLAY_[TYPE]_impl.h` where `[TYPE]` depends on the display configuration used.

8.6.2 Custom

This implementation of the MicroUI `Display` API targets custom display (see “Display Configurations”. It provides some low level API. The BSP has to implement these LLAPI making the link between the MicroUI C library `display` and the BSP display driver.

The LLAPI to implement are listed in the header file `LLDISPLAY_CUSTOM_impl.h`.

8.6.3 Handy Guide

The following flow chart provides a handy guide to selecting the appropriate buffer mode according to the hardware configuration.

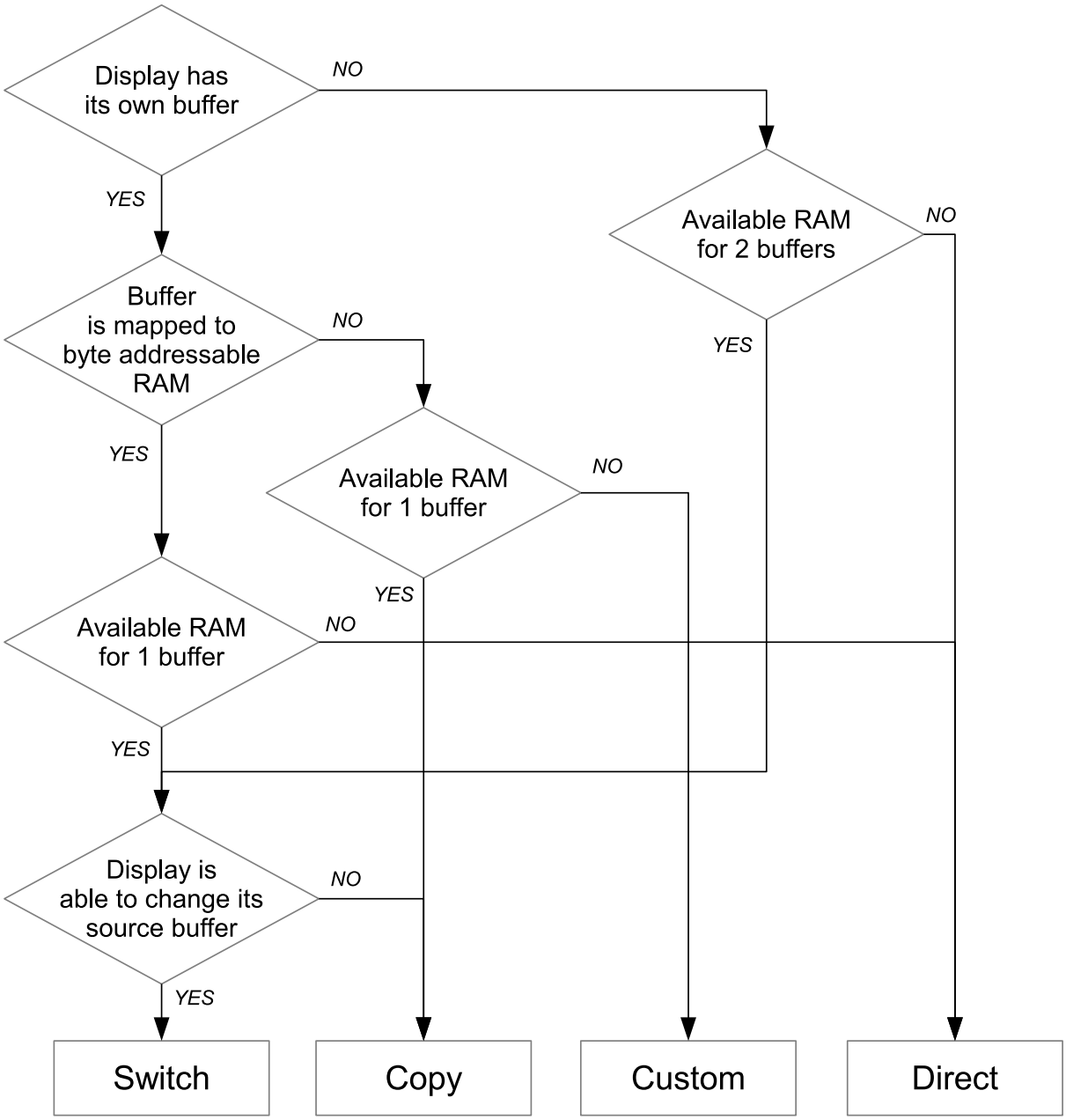


Figure 8.3. Buffer Modes

8.7 Low Level API

8.7.1 Implementation

The display Low Level API is designed to allow multiple implementations and instances (see Low Level API Pattern in [CM3REF]). Each display implementation defines a data structure, and functions take an instance of this data structure as the first parameter.

The following illustration shows part of the content of the files for a display driver.

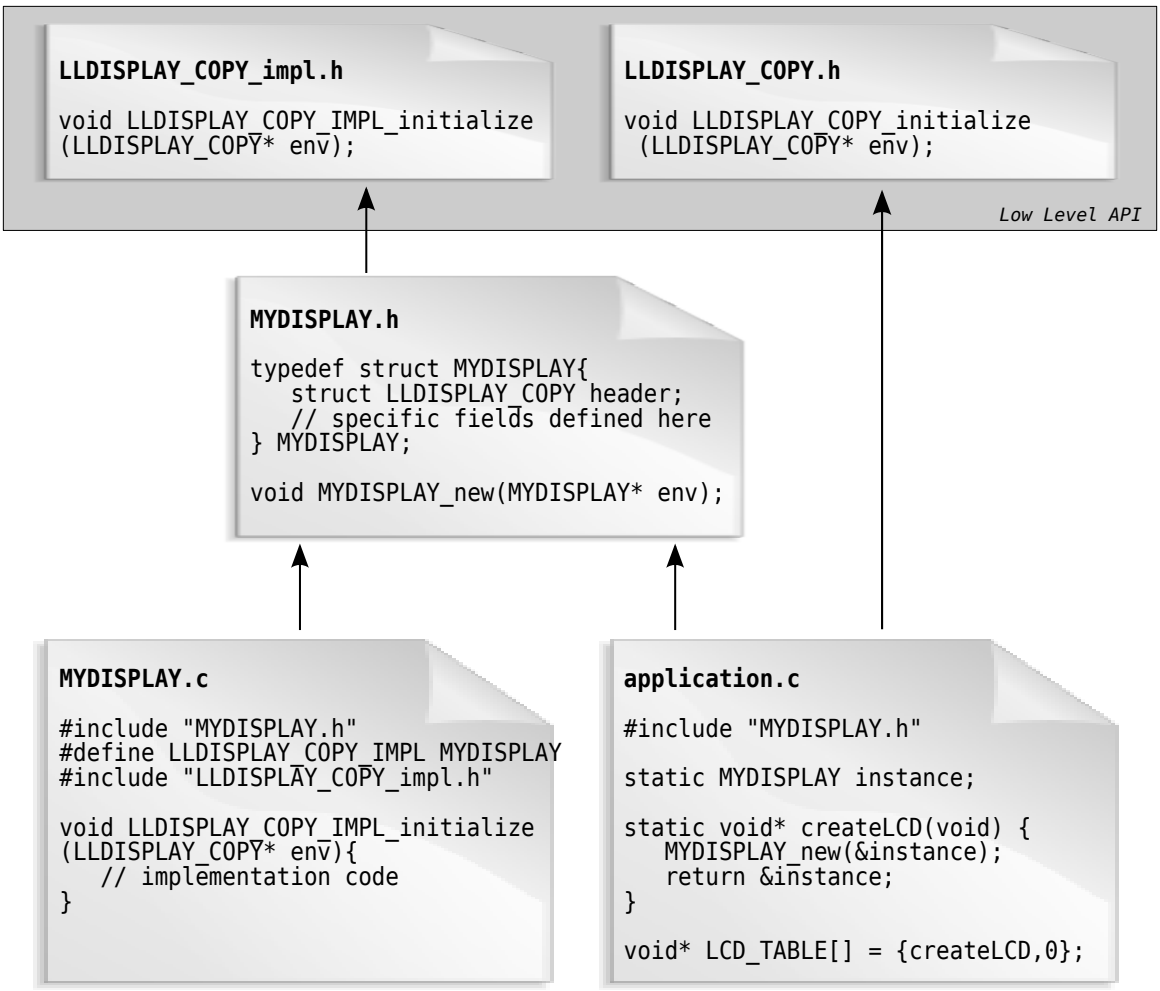


Figure 8.4. Display Driver Example

The name of the implementation must be set at the top of the C file `MYDISPLAY.c`:

```
#define LLDISPLAY_COPY_IMPL MYDISPLAY
```

This defines the name of this implementation of the `LLDISPLAY_COPY` interface to be `MYDISPLAY`.

The data structure managed by this implementation is defined in `MYDISPLAY.h`. In this example the structure contains only the default data, in the header field. Note that the header must be the first field in the structure. The name of this structure must be the same as the implementation name (`MYDISPLAY` in this example).

The header file `MYDISPLAY.h` must also declare the "new" function used to initialize display instances. The name of this function must be the implementation name with `_new` appended, and it takes as its sole argument a pointer to an instance of the display data structure.

The driver needs to implement the functions specified in the `LLDISPLAY_COPY_impl.h` file.

The display module needs to know, when the Java application is built, the name of the implementation. This mapping is defined in the BSP definition file (see [USRMANUAL]). In this example the `bsp.xml` file would contain:

```
<bsp>
    <nativeImplementation
        name="MYDISPLAY"
        nativeName="LLDISPLAY_COPY"
    />
</bsp>
```

Figure 8.5. Display Driver Declaration

where `nativeName` is the name of the interface, and `name` is the name of the implementation.

8.7.2 Instance Creation

The display library must be told about the driver instances. The implementation must declare a global data table, named `LCD_TABLE`, which contains functions that create the display driver instances. The table must be `null` terminated. The create function must call the "new" function and return the address of the data structure.

8.8 Dependencies

- MicroUI module (see “MicroUI”)
- `LLDISPLAY_[TYPE]_impl.h` implementation if standard or custom implementation is chosen (see “Implementations”).

8.9 Installation

Display is a sub-part of MicroUI Java library. When MicroUI module is installed, the Display module should be installed too, otherwise it may throw linker errors during Java application compilation.

In the platform configuration file, check `UI > Display` to install it. When checked, the properties file `display > display .properties` is required during platform creation to configure the module. This configuration step is used to choose the kind of implementation (see “Implementations”).

The properties file have to / can contain the following properties:

- `type` [optional, default value is "default"]: Defines the type of display implementation. Two types are available. All others types are forbidden (throw a generation error). Note: "`default`" type requires 3rd-party properties to configure the display module (see after).
  - `default`: select this configuration to use a standard display implementation (for standard display devices).
  - `custom`: select this configuration to use a custom display implementation (for non standard display devices).
- `bpp` [optional, default value is "" (*empty*)]: Only useful when '`type`' == '`default`'. Defines the number of bits per pixels the display device is using to render a pixel. Expected value is one among this list: 1 | 2 | 4 | 8 | 16 | 32. All others values are forbidden (throw a generation error).
- `layout` [optional, default value is "" (*empty*)]: Only useful when '`type`' == '`default`'. Defines the pixels memory layout the display device is using. Two modes are available:
  - `xy`: the first memory address targets the pixel (0,0), next memory address (according BPP) targets the pixel (1,0)
  - `yx`: the first memory address targets the pixel (0,0), next memory address (according BPP) targets the pixel (0,1)

  All others modes are forbidden (throw a generation error)

- `mode` [optional, default value is "" (*empty*)]: Only useful when '`type`' == '`default`'. Defines the buffering mode (simple or double). Standard available modes are:
  - `direct`: simple buffering mode
  - `copy`: double buffering mode (display holds its own memory space)
  - `switch`: double buffering mode (display is able to change the memory buffer it targets at runtime)
  - `hwlayers`: copy mode + specific LLAPI to use the hardware features in order to implement the MicroUI Layers API

8.10 Use

The MicroUI Display APIs are available in the class `ej.microui.io.Display`.

9 Audio

9.1 Introduction

The Audio module contains the C part of the MicroUI implementation which manages audio devices. This module is composed of the C part of MicroUI Audio API (a built-in C archive).

This MicroUI feature is not available on this product. It is not possible to play any tone. Only a *stub* implementation of C library is available. This C library must be linked by the third-party C IDE when MicroUI module is installed in the Java platform.

9.2 Dependencies

- MicroUI module (see “MicroUI”)

9.3 Installation

Audio is a sub-part of MicroUI Java library. When MicroUI module is installed, the Audio module should be installed too, otherwise it may throw linker errors during Java application compilation.

10 Alpha Numeric Display

10.1 Introduction

The AlphaNumeric module contains the C part of the MicroUI implementation which manages alpha numeric displays. This module is composed of the C part of MicroUI AlphaNumeric API (a built-in C archive).

This MicroUI feature is not available on this product. It is not possible to drive any alpha numeric display. Only a *stub* implementation of C library is available. This C library must be linked by the third-party C IDE when MicroUI module is installed in the Java platform.

10.2 Dependencies

- MicroUI module (see “MicroUI”)

10.3 Installation

AlphaNumeric is a sub-part of MicroUI Java library. When MicroUI module is installed, the AlphaNumeric module should be installed too, otherwise it may throw linker errors during Java application compilation.

11 Images

The Image Engine is composed of:

- The "Image Engine Core" module which is able to load and drawing simultaneously some pre-generated images and some dynamic images.
- An "Image Generator" module, for converting standard image formats into the display image format before runtime (pre-generated images).
- A set of "Image Decoder" modules, for converting standard image formats into the display image format at runtime. Each Image Decoder is an additional module of the main module "Image Engine".

11.1 Image Engine Core

11.1.1 Introduction

The Image Engine Core module is a built-in module of the MicroUI module (see “MicroUI”) for the Java side and of the Display module (see “Display”) for the C side.

11.1.2 Functional Description

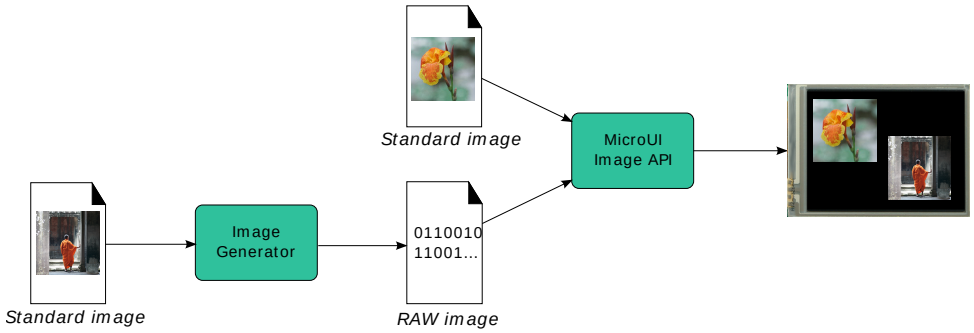


Figure 11.1. Image Engine Core Principle

Process overview:

1. The user specifies the pre-generated images to embed (see “Image Generator”) and / or the images to embed as regular Java resources (see “Image Decoder”)
2. The files are embedded as resources with the Java application. The files data are linked into the FLASH memory.
3. When the Java application creates a MicroUI Image object, the Image Engine Core loads the image calling the right sub Image Engine module (see “Image Generator” and “Image Decoder”) to decode the specified image.
4. When the Java application draws this MicroUI Image on the display (or on another image) the decoded image data is used, and no more decoding is required, so the decoding is done only once.
5. When the MicroUI Image is no longer needed, it is garbage collected by the platform, the Image Engine Core asks to the right sub Image Engine module (see “Image Generator” and “Image Decoder”) to free the image working area.

11.1.3 Image Identifier

Before loading an image calling the right sub module, the Image Engine Core module tries first to attribute an unique identifier to the future decoded image. This identifier will be used to retrieve the image after the decoding step in order to draw it and to free it.

This identifier targets too some meta data for the image (same size for all images, specific to the Display module). An identifier is reserved to an image as longer as the image is used by the Java application. When the MicroUI Image is no longer needed, it is garbage collected by the platform. The identifier (and its meta data memory space) is freed. A new image is so able to use this identifier.

To prevent some C allocation at runtime, the number of identifiers and the memory space useful to store the image meta data are allocated at compile-time. By consequence the available number of identifiers is limited. The MicroEJ launcher of the Java application has to specify the number of identifiers (refer to the chapter “Launch Options” (`Target` > `Memory`) to have more information where specify this number of identifiers.)

When the limit of identifiers is reached, the MicroUI library throws an `OutOfMemoryError`, error code `-5`. In this case try to augment the number of concurrent images in the MicroEJ launcher or try to remove the links on useless MicroUI Image objects.

11.1.4 Dependencies

- MicroUI module (see “MicroUI”)
- Display module (see “Display”)

11.1.5 Installation

Image Engine Core modules is a part of MicroUI module and Display module. Install them to be able to use some images.

11.1.6 Use

The MicroUI image APIs are available in the class `ej.microui.io.Image`.

11.2 Image Generator

11.2.1 Introduction

The Image Generator module is an off-board tool that generates image data ready to be displayed without needing additional runtime memory. The two main advantages are:

- A pre-generated image is already encoded in the format of the display implementation. The image loading is very fast and does not require any RAM memory.
- No extra support is needed (no runtime decoder).

11.2.2 Functional Description

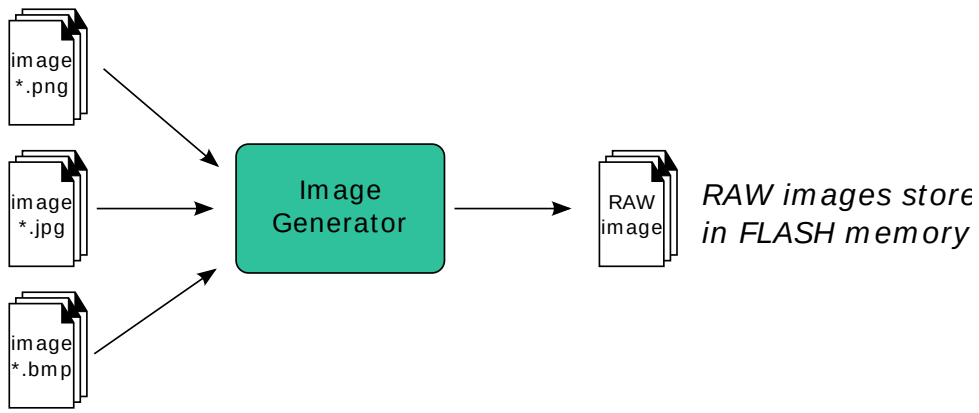


Figure 11.2. Image Generator Principle

Process overview (see too “Functional Description”)

1. The user defines, in a text file, the images to load.

2. The Image Generator outputs a raw file for each image to convert (the raw format is display device dependent).

3. The raw files are embedded as (hidden) resources within the Java application. The raw files data are linked into the FLASH memory.

4. When the Java application creates a MicroUI Image object which targets a pre-generated image, the Image Engine Core has only to do a link from the MicroUI image object to the data in the FLASH memory. Therefore, the loading is very fast; only the image data from the FLASH memory is used: no copy of the image data is done to the RAM memory first.

5. When the MicroUI Image is no longer needed, it is garbage collected by the platform, which just deletes the useless link to the FLASH memory.

11.2.3 Configuration File

The Image Generator uses a configuration file (also called the "list file") for describing images that need to be processed. The list file is a text file where each line describes an image to convert. The image is described as a resource path and should be available from the application classpath.

Each line can add optional parameters (separated by a ':') which define and/or describe the output file format (raw format). When no option is specified the image is converted into the default format.

Below is an example of a list file for the Image Generator:

```
image1
image2:BITMAP
```

Figure 11.3. Image Generator Configuration File Example

Refer to the chapter “Launch Options” (`Libraries` > `MicroUI` > `Image`) to have more information where specify this images configuration file.

11.2.4 Extensions Purpose

The output representation of the images is dependent on the drivers that run the underlying screen. Indeed, the output raw format is specific to each display device. The Image Generator tool provided is expandable by extensions, each extension implementing a dedicated display device layout.

The Display module provides pre-configured display implementation with specific layouts (see “Display”). For each one, a dedicated extension interface is also provided within the Image Generator tool.

11.2.5 Custom Extensions

A custom extension implementation has the responsibility to encode an image into a raw format that matches the display driver's display buffer encodings. Implement a custom extension as a subclass of the abstract class CustomDisplayExtension [3] . This extension must be used when the Display module implementation is *custom* (see “Display”).

- `getID` method returns the unique identifier for this extension, which identifies the extension-driver pair.

- `canEncodeImage` method returns true when the extension implementation is able to encode the given image. If not, a warning will be thrown during the images pre-generation.

[3]Package com.is2t.microej.microui.image

- `encodeImage` method encodes the image and returns an encoded byte array. This sequence of bytes is intended to be understood by the `LLDISPLAY_CUSTOM_IMPL_copyArea` when the id of an image decoded by the image generator is passed as `srcBufferId`.

Note that the output(s) format(s) name(s) and potential options are user specific.

11.2.6 Generic Extension

The Display module provides generic display implementation for several combinations between the number of bits-per-pixels (1, 2, 4, 8, 16 and 32) and for several memory layouts (line-per-line and column-per-column). The Image Generator tool provides a simple extension to implement to target these kinds of displays: `GenericDisplayExtension` [4].

Two methods must be implemented in relation to the way the driver has built the layout of the display buffers in memory:

- `getID` method returns the unique identifier for this extension. For devices that have only a single graphical display (therefore only one display memory layout), this may always return 0. Otherwise, it should characterize the screen driver using this ID.

- `convertRGBColorToDisplayColor` method is used to convert a 32-bits ARGB color into the display pixel memory representation.

Note: the Image Generator automatically uses the right number of bits to represent a pixel (BPP) and respect the memory buffer layout using the result of the installation of the Display module.

This extension provides two output formats. `BITMAP` and `RLE1`. First one is the default one and the second is only available for the 16 bits-per-pixels displays.

11.2.7 Create an Extension

Follow the steps below for each display available on the platform:

1. First of all create a new J2SE project, called, for example, `imageGeneratorExtension`.

2. In the project's Java build path (project's property window, select **Java Build Path** → **Libraries tab**), add the variable `IMAGE-GENERATOR-x.y`.

3. Create the package `com.is2t.microui.generators.extension`.

4. Create a class in the package whose name must be: `MicroUIGeneratorExtension`.

5. The Java class must implement the extension interface available in the library `IMAGE-GENERATOR-x.y` (see previous chapters). Fill the required methods.

The Java project should look like this:

Figure 11.4. Image Generator Extension Project

[4]Package com.is2t.microej.microui.image

With a Java class like this:

```
package com.is2t.microui.generators.extension;

import com.is2t.microej.microui.image.GenericDisplayExtension;

public class MicroUIGeneratorExtensionMyLCD implements GenericDisplayExtension{

    public int convertRGBColorToDisplayColor(int rgbColor) {
        return (char)
            ((rgbColor & 0xf80000) >>> 8) |
            ((rgbColor & 0x00fc00) >>> 5) |
            ((rgbColor & 0x0000f8) >>> 3);
    }

    public int getID() {
        return 0;
    }
}
```

Figure 11.5. Image Generator Extension Implementation Example

11.2.8 BITMAP Output Format

The default embedded image data format provided by the Image Generator tool when using a generic extension is to encode the image into the exact display memory representation. If the image to encode contains some transparent pixels the output file will embed the transparency according the display implementation capacity. When all pixels are fully opaque, no extra information will be stored in output file to win some memory space.

Advantages:

- drawing an image is very fast.
- supports alpha encoding.

Disadvantages:

- no compression: the image size in bytes is proportional to the number of pixels.

The two lines below describe the same encoding, because the default options are used.

```
image1
image2:BITMAP
```

Figure 11.6. BITMAP Output Format Examples

11.2.9 RLE1 Output Format

The image engine can display embedded images that are encoded into a compressed format which encodes several consecutive pixels into one or more 16-bits words.

This encoding is reserved for generic extension which targets a 16-bits encoding (BPP = 16) and a maximum alpha level of 2 (alpha level is always assumed to be 2, even if the image is not transparent).

- Several consecutive pixels have the same color (2 words).
- first 16-bits word specifies how many consecutive pixels have the same color.
- second 16-bits word is the pixels' color.
- Several consecutive pixels have their own color (1 + n words).

- first 16-bits word specifies how many consecutive pixels have their own color.
- next 16-bits words is the next pixel color.
- Several consecutive pixels are transparent (1 word).
- 16-bits word specifies how many consecutive pixels are transparent.

Advantages:

- supports 0 & 2 alpha encoding.
- good compression when several consecutive pixels respect one of the three previous rules.

Disadvantages:

- drawing an image is slightly slower than BITMAP format.

11.2.10 Dependencies

- Image Engine Core module (see “Image Engine Core”)
- Display module (see “Display”): this module gives the characteristics of the graphical display useful to configure the Image Generator.

11.2.11 Installation

Image Generator is an additional module for MicroUI Java library. When MicroUI module is installed, install this module to be able to target pre-generated images.

In the platform configuration file, check `UI > Image Generator` to install it. When checked, the properties file `imageGenerator > imageGenerator .properties` is required during platform creation to configure the module. This configuration step is used to identify the extension class name (see “Create an Extension”).

The properties file have to / can contain the following properties:

- `project.name` [mandatory]: Defines the name image generator extension project. This project must be available in same workspace as the current project. It must contains a source folder called "src". This source folder must contain the Java class "com.is2t.microui.generators.extension.MicroUIGeneratorExtension".

11.2.12 Use

The MicroUI Image APIs are available in the class `ej.microui.io.Image`. There is no specific APIs to use a pre-generated image. When an image has been pre-processed, the MicroUI Image APIs `createImage*` will load this image.

11.3 Image Decoder

11.3.1 Introduction

The Image Engine provides runtime decoders which allow the dynamic loading of images without using the Image Generator (see “Image Generator”). The two main advantages are:

- The original image is embedded as a resource with the Java application.
- The original image size in bytes is often smaller than a pre-generated image (especially in PNG mode).

11.3.2 Functional Description

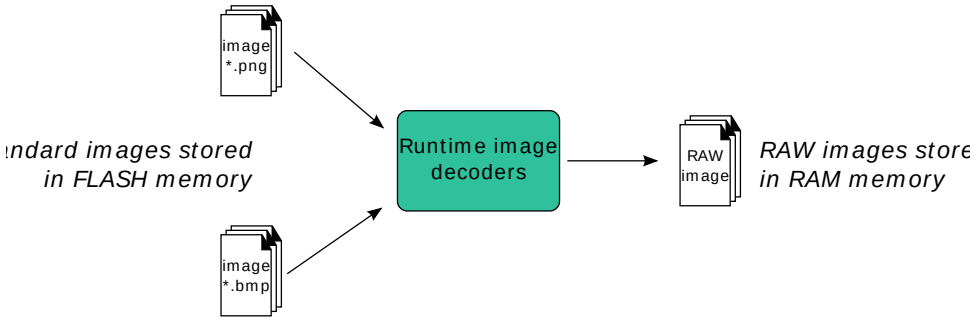


Figure 11.7. Image Decoder Principle

Process overview (see too “Functional Description”)

1. The user specifies the images to embed as regular Java resources.

2. The original files are embedded as resources with the Java application. The original files data are linked into the FLASH memory.

3. When the Image Engine Core calls the decoder to load an image, it transforms the image into a raw format compatible with the display format. Then it may need some additional RAM memory to store some working buffers. At the end of the decoding step the working buffers are freed: only the decoded image memory needs to be held onto.

4. When the Image Engine Core calls the decoder to free the image resources, the decoder frees the decoded image buffer area.

11.3.3 Decoders

The UI extension provides two Image Decoders modules:

- PNG Decoder: is a full PNG decoder that implements the PNG format (www.w3.org/Graphics/PNG [http://www.w3.org/Graphics/PNG] .). Regular, interlaced, indexed (palette) compressions are handled. The RAM used by the decoder is allocated outside the Java heap.

- BMP Monochrome Decoder: .bmp format files that embed only 1 bit per pixel can be decoded by this decoder. The RAM used by the decoder to store the decoded image is outside the Java heap.

11.3.4 Dependencies

- Image Engine Core module (see “Image Engine Core”)

11.3.5 Installation

The Image Decoders modules are some additional modules to the Display module. The decoders belong to distinct modules, and either or several may be installed.

In the platform configuration file, check `UI > Image PNG Decoder` to install The runtime PNG decoder. Check `UI > Image BMP Monochrom Decoder` to install The runtime BMP monochrom decoder.

Once installed a decoder is not automatically embedded in final application ELF file. A data table named `IMAGE_DECODERS_TABLE` (null terminated) has to be defined to allow to the Display module to retrieve the list of available decoders. Each decoder provides an header file useful to instantiate itself in C world. An example of the declaration is:

```
#include "LLDISPLAY_DECODER_PNG.h"
#include "LLDISPLAY_DECODER_BMPM.h"

void* IMAGE_DECODERS_TABLE[] = {
    LLDISPLAY_DECODER_PNG_createPNGImageDecoder,
    LLDISPLAY_DECODER_BMPM_createMonochromBMPImageDecoder,
    0
};
```

Figure 11.8. Image Decoders Table

11.3.6 Use

The MicroUI Image APIs are available in the class `ej.microui.io.Image`. There is no specific APIs to use a runtime image. When an image has not been pre-processed (see “Image Generator”), the MicroUI Image APIs `createImage*` will load this image.

12 Fonts

The Font Engine is composed of:

- The "Font Engine Core" module which decodes and prints at application runtime the platform-dependent fonts files generated by the "Font Generator".
- A "Font Designer" module: graphical tool which runs within the MicroEJ workbench used to build and edit MicroUI fonts; it stores fonts in a platform-independent format.
- A "Font Generator" module, for converting fonts from the platform-independent format into a platform-dependent format.

The three modules are complementary: a MicroUI font must be created and edited with the Font Designer before being integrated as a resource thanks the Font Generator. Finally the Font Engine Core uses the generated fonts at runtime.

The Font Designer module and Font Generator module options are directly the consequence of the Font Engine Core capacities. Understand the Font Engine Core capacities is so required to use correctly the modules.

12.1 Font Engine Core

12.1.1 Introduction

The Font Engine Core module is a built-in module of the MicroUI module (see “MicroUI”) for the Java side and of the Display module (see “Display”) for the C side.

12.1.2 Functional Description

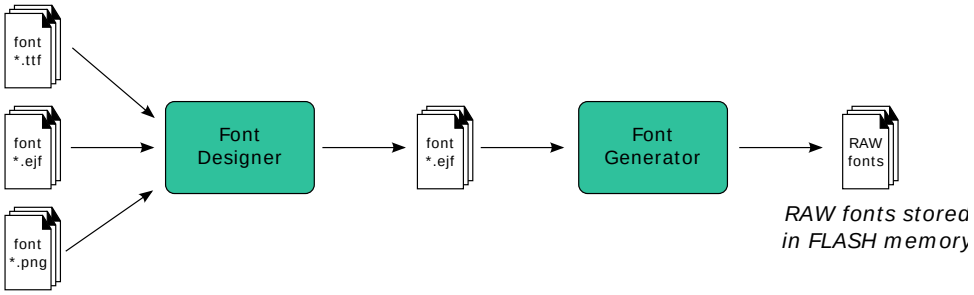


Figure 12.1. Font Generation

Process overview:

1. User uses the Font Designer module to create a new font and imports characters from system fonts (`*.ttf` files) and / or user images (`*.png`, `*.jpg`, `*.bmp` etc.).
2. Font Designer module saves the font as a MicroEJ Font (`*.ejf` file).
3. The user defines, in a text file, the fonts to load.
4. The Font Generator outputs a raw file for each font to convert (the raw format is display device dependent).
5. The raw files are embedded as (hidden) resources within the Java application. The raw files data are linked into the FLASH memory.
6. When the Java application creates a MicroUI DisplayFont object which targets a pre-generated image, the Font Engine Core has only to do a link from the MicroUI DisplayFont object to the data in the FLASH memory. Therefore, the loading is very fast; only the font data from the FLASH memory is used: no copy of the image data is done to the RAM memory first.
7. When the MicroUI DisplayFont is no longer needed, it is garbage collected by the platform, which just deletes the useless link to the FLASH memory.

12.1.3 Font Format

The font engine module provides fonts that conform to the Unicode Standard [U61]. `.ejf` files hold font properties:

- Identifiers: fonts hold at least one identifier that can be one of the predefined Unicode scripts [U61] or a user-specified identifier. The intention is that an identifier indicates that the font contains a specific set of character codes, but this is not enforced.
- Font height and width, in pixels. A font has a fixed height. This height includes the white pixels at the top and at the bottom of each character, simulating line spacing in paragraphs. A monospace font is a font where all characters have the same width. For instance, a '!' representation has the same width as a 'w'. In a proportional font, 'w' will be wider than a '!'. No width is specified for a proportional font.

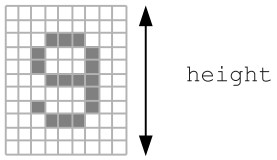


Figure 12.2. Font height

- Baseline, in pixels: all characters have the same baseline, which is an imaginary line on top of which the characters seem to stand. Characters can be partly under the line, for example 'g' or '}'. The number of pixels specified is the number of pixels above the baseline.

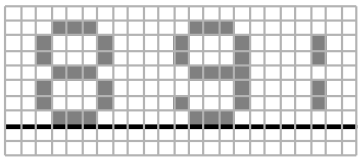


Figure 12.3. Font baseline

- Space character size, in pixels: for proportional fonts, the Space character (`0x20`) is a specific character because it has no filled pixels, and so its width must be specified. For monospace, the space size is equal to the font width (and hence the same as all other characters).
- Styles: a font holds either a combination of these styles: BOLD, ITALIC, UNDERLINED, or is said to be PLAIN.
- Runtime filters: some fonts may allow the font engine to apply a transformation (i.e. a filter) on characters before they are displayed in order to provide some visual effect on characters (BOLD, ITALIC, UNDERLINED). Unless specified, a font allows the font engine to apply any of its filters.
- When the selected font does not have a graphical representation of the required character, a rectangle is displayed instead. For proportional fonts, the width is one third of the height of the font.

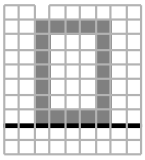


Figure 12.4. Default character

12.1.4 Font Selection

The font engine implements the [MUI] selection semantics, and also tries to select fonts for which styles are built-in instead of applying a runtime filter. The selection of the font is done based on the following process:

1. select fonts that define the specified identifier.

2. select within the step1 fonts, those whose height is the closest to the specified height.
3. select within the step2 fonts, those with built-in styles that match the specified styles.
4. if more than one font is selected by the steps above, select those fonts that have the most build-in styles. If there is still more than one font, one is selected arbitrarily.

12.1.5 Runtime Transformation: Filters

The user interface extension font engine provides 3 runtime filters that may apply if the (current selected) font allows it. The filters are:

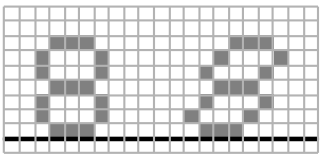
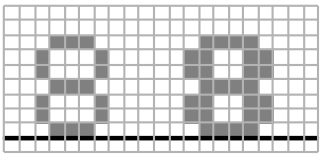
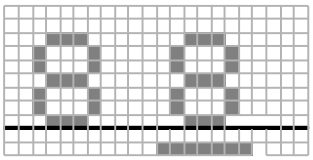
Name	Description	Rendering sample
ITALIC	Pixels on upper rows are shifted right. The higher a pixel is relative to the base line, the more it is right-shifted.	
BOLD	1 pixel is added to the right of each original pixel.	
UNDERLINED	A line is displayed two pixels below the base-line position.	

Table 12.1. The 3 font runtime style transformations (filters).

Multiple filters may apply at the same time, combining their transformations on the displayed characters.

12.1.6 Pixel Transparency

The font engine renders the font according the the value stored for each pixel. If the value is 0, the pixel is not rendered. If the value is the maximum value (e.g. the value 3 for 2 bits-per-pixel), the pixel is rendered using the current foreground color, completely overwriting the current value of the destination pixel. For other values, the pixel is rendered by blending the selected foreground color with the current color of the destination.

If n is the number of bits-per-pixel, then the maximum value of a pixel (pmax) is 2^n – 1. The value of each color component of the final pixel is equal to:

foreground * pixelValue / pmax + background * (pmax - pixelValue) / pmax + adjustment

where adjustment is an adjustment factor specified in the board support package of the platform.

12.1.7 Font Identifier

All fonts are loaded at MicroUI startup. Before loading a font, the Image Engine Core module tries first to attribute an unique identifier to the future loaded font. This identifier will be used to retrieve the font after the loading step in order to draw it and to free it.

An identifier targets a font file (ejf raw file), which can contains until eight DisplayFont inside. To prevent some C allocation at runtime, the number of identifiers is allocated at compile-time. By consequence the available number of identifiers is limited. The MicroEJ launcher of the Java application has to specify the number of identifiers (refer to the chapter “Launch Options” (`Target` > `Memory`) to have more information where specify this number of identifiers.)

Note that this number has to include the number of system fonts. A system font is a font file specified during the MicroUI static initialization step (see “Static Initialization”).

When the limit of identifiers is reached, the MicroUI library throws an error and the non-loaded fonts are unusable.

12.1.8 Dependencies

- MicroUI module (see “MicroUI”)
- Display module (see “Display”)

12.1.9 Installation

Font Engine Core modules is a part of MicroUI module and Display module. Install them to be able to use some fonts.

12.1.10 Use

The MicroUI font APIs are available in the class `ej.microui.io.DisplayFont`.

12.2 Font Designer

12.2.1 Introduction

The Font Designer module is a graphical tool (Eclipse plugin) running within the MicroEJ workbench used to build and edit MicroUI fonts. It stores fonts in a platform-independent format.

12.2.2 Functional Description

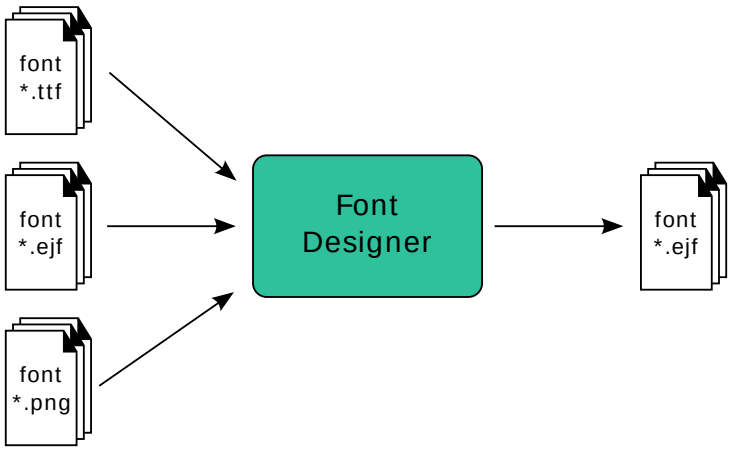


Figure 12.5. Font Generation

12.2.3 Create a MicroEJ Font

To create a MicroEJ font, follow the steps below:

1. Open the Eclipse wizard: **File** → **New** → **Other** → **MicroEJ** → **MicroEJ Font**.
2. Select a directory and a name.
3. Click **Finish**.

Once the font is created, a new editor is opened: the MicroEJ Font Designer Editor.

12.2.4 Edit a MicroEJ Font

You can edit your font with the MicroEJ Font Designer Editor (by double-clicking on a `*.ejf` file or after running the new MicroEJ Font wizard).

This editor is divided into three main parts:

- The top left part manages the main font properties.

- The top right part manages the character to embed in your font.
- The bottom part allows you to edit a set of characters or an individual character.

12.2.4.1 Main Properties

The main font properties are:

- font size: height and width (in pixels).
- baseline (in pixels).
- space character size (in pixels).
- styles and filters.
- identifiers.

Refer to the following sections for more information about these properties.

12.2.4.1.1 Font Height

A font has a fixed height. This height includes the white pixels at the top and at the bottom of each character simulating line spacing in paragraphs.

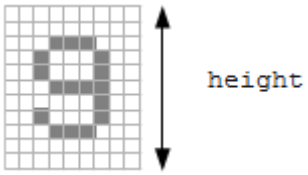


Figure 12.6. Font Height

12.2.4.1.2 Font Width: Proportional and Monospace Fonts

A monospace font is a font in which all characters have the same width. For instance, a '!' representation will be the same width as a 'w' (they will be in the same size rectangle of pixels). In a proportional font, a 'w' will be wider than a '!'.

A monospace font usually offers a smaller memory footprint than a proportional font because the Font Designer does not need to store the size of each character. As a result, this option can be useful if the difference between the size of the smallest character and the biggest one is small.

12.2.4.1.3 Baseline

Characters have a baseline: an imaginary line on top of which the characters seem to stand. Note that characters can be partly under the line, for example, 'g' or '}'.

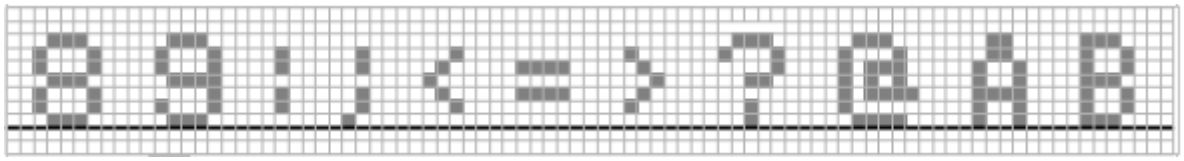


Figure 12.7. The Baseline

12.2.4.1.4 Space Character

The Space character (0x20) is a specific character because it has no filled pixels. From the Main Properties Menu it is possible to fix the space character size in pixels.

Note: if the font is monospace, the space size is equal to font width.

12.2.4.1.5 Styles and Filters

A MicroUI font holds a style: PLAIN, BOLD, ITALIC, UNDERLINED and the combinations between BOLD, ITALIC and UNDERLINED. Font Designer can use one file to describe several MicroUI fonts.

For instance, a font file that describes a PLAIN font can also describe an UNDERLINED font because the MicroUI implementation just has to draw a line under the characters. In this way, from a developer's point of view, there are two fonts: a PLAIN font and an UNDERLINED font. From the Font Designer point of view, there are also two fonts but they use the same data file. Font Designer adds a tag to describe the UNDERLINED font in the generated font file.

This tag is a *filter*. When a file contains one or more filters, MicroUI implementation knows that it has to perform post processing to obtain a specific MicroUI font from the encoded font.

Alternatively, the user can create two distinct files to describe the two fonts. From the MicroUI application point of view, there are always two fonts: a PLAIN font and an UNDERLINED font, but no post processing step is required (no filter tag).

Examples:

1. A font file contains the styles PLAIN and UNDERLINED and the filters PLAIN and UNDERLINED. The MicroUI implementation detects two MicroUI fonts. To draw each font, the PLAIN filter or the UNDERLINED filter are used accordingly.

2. A font file contains the styles PLAIN and UNDERLINED and the filter PLAIN. The MicroUI implementation detects two MicroUI fonts. To draw the underlined font, it will not apply the underlining process (the filter UNDERLINED is absent). So the MicroUI underlined font will have the same rendering as the MicroUI plain font.

Font Designer features three drop-downs, one for each of BOLD, ITALIC and UNDERLINED. Each drop down has three options:

- None – Font Designer will not set this style nor include a filter for it.
- Built-in – Font Designer will set this style, but not include a filter for it.
- Dynamic – Font Designer will set this style, and include a filter for it.

If all three drop-downs are set to None, only a plain font is generated.

The number of fonts that will result is shown below the drop-downs.

12.2.4.1.6 Identifiers

A number of identifiers can be attached to a MicroUI font. At least one identifier is required to specify the font. Identifiers are a mechanism for specifying the content of the font – the set or sets of characters it contains. The identifier may be a standard identifier (for example, LATIN) or a user-defined identifier. Identifiers are numbers, but standard identifiers, which are in the range 0 to 80, are typically associated with a handy name. A user-defined identifier is an identifier with a value of 81 or higher.

12.2.4.2 Character List

The list of characters can be populated through the **import** button which allows you to import characters from system fonts, images or another MicroEJ font.

12.2.4.2.1 Import from System Font

This page allows the selection of the system font to use (left part) and the range of characters. There are predefined ranges of characters below the font selection and a custom selection picker (e.g. 0x21 to 0xfe for Latin characters).

The right part displays the selected characters with the selected font. If the background color of a displayed character is red, it means that the character is too large for the defined height, or in case of a monospace font, if it is too high or too wide. You can then adjust the font properties (font size and style) to ensure that characters will not be truncated.

When your selection is done, click the **Finish** button to import this selection into your font.

12.2.4.2.2 Import from Images

This page allows the loading of images from a directory. The images must be named as follows: `0x[UTF-8].[extension]`.

When your selection is done, click the **Finish** button to import the images into your font.

12.2.4.3 Character Editor

When a single character is selected in the list, the character editor is opened.

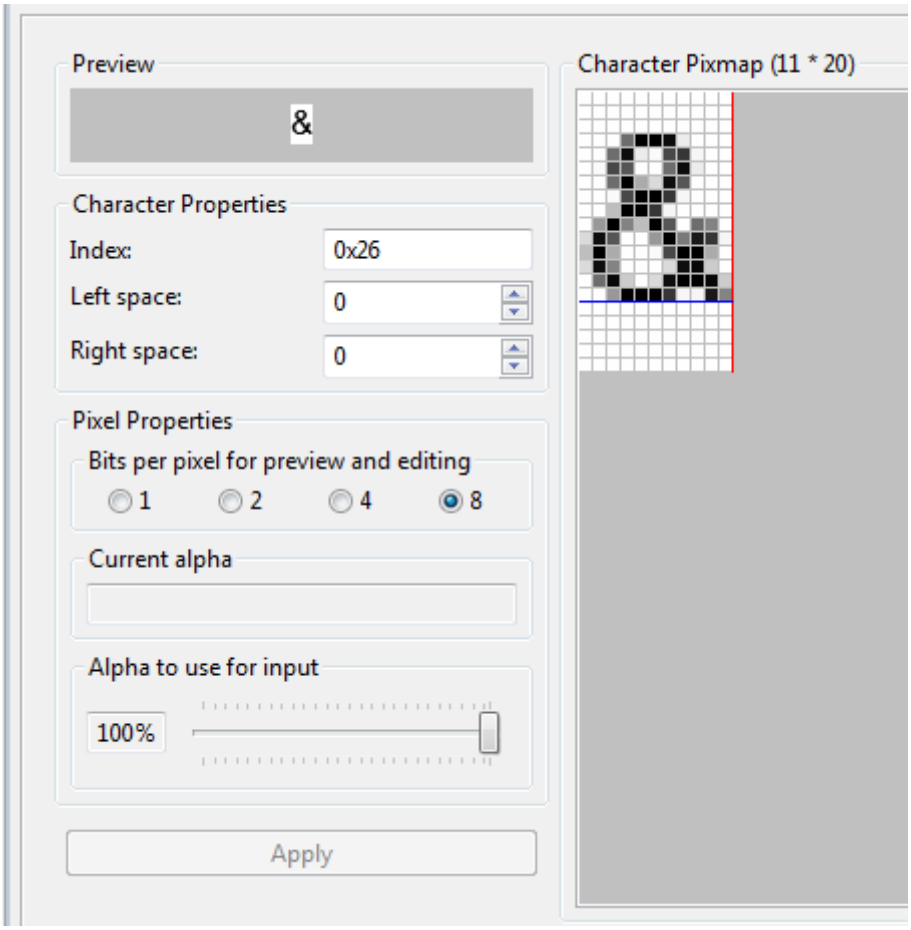


Figure 12.8. Character Editor

You can define specific properties such as left and right space, or index. You can also draw the character pixel by pixel - a left-click in the grid draws the pixel, a right-click erases it.

The changes are not saved until you click the **Apply** button. When changes are applied to a character, the editor shows that the font has changed, so you can now save it.

The same part of the editor is also used to edit a set of characters selected in the top right list. You can then edit the common editable properties (left and right space) for all those characters at the same time.

12.2.4.3.1 Working With Anti-Aliased Fonts

By default, when characters are imported from a system font each pixel is either fully opaque or fully transparent. Fully opaque pixels show as black squares in the character grid in the right-hand part of the character editor; fully transparent pixels show as white squares.

However, the pixels stored in an `ejf` file can take one of 256 grayscale values. A fully-transparent pixel has the value 255 (the RGB value for white), and a fully-opaque pixel has the value 0 (the RGB value for black). These grayscale values are shown in parentheses at the end of the text in the **Current alpha**

field when the mouse cursor hovers over a pixel in the grid. That field also shows the transparency level of the pixel, as a percentage, where 100% means fully opaque.

It is possible to achieve better-looking characters by using a combination of fully-opaque and partially-transparent pixels. This technique is called *anti-aliasing*. Anti-aliased characters can be imported from system fonts by checking the **anti aliasing** box in the import dialog. The '&' character shown in the screenshot above was imported using anti aliasing, and you can see the various gray levels of the pixels.

When the Font Generator converts an `ejf` file into the raw format used at runtime it can create fonts with characters that have 1, 2, 4 or 8 bits-per-pixel (bpp). If the raw font has 8 bpp, then no conversion is necessary and the characters will render with the same quality as seen in the character editor. However, if the raw font has less than 8 bpp (the default is 1 bpp) any gray pixels in the input file are compressed to fit, and the final rendering will be of lower quality (but less memory will be required to hold the font).

It is useful to be able to see the effects of this compression, so the character editor provides radio buttons that allow the user to preview the character at 1, 2, 4, or 8 bpp. Furthermore, when 2, 4 or 8 bpp is selected, a slider allows the user to select the transparency level of the pixels drawn when the left mouse button is clicked in the grid.

12.2.4.4 Previewing a Font

You can preview your font by pressing the **Preview...** button which opens the Preview wizard. In the Preview wizard, press the **Select File** button and select a text file which contains text that you want to see rendered using your font. Characters that are in the selected text file but not available in the font will be shown as red rectangles.

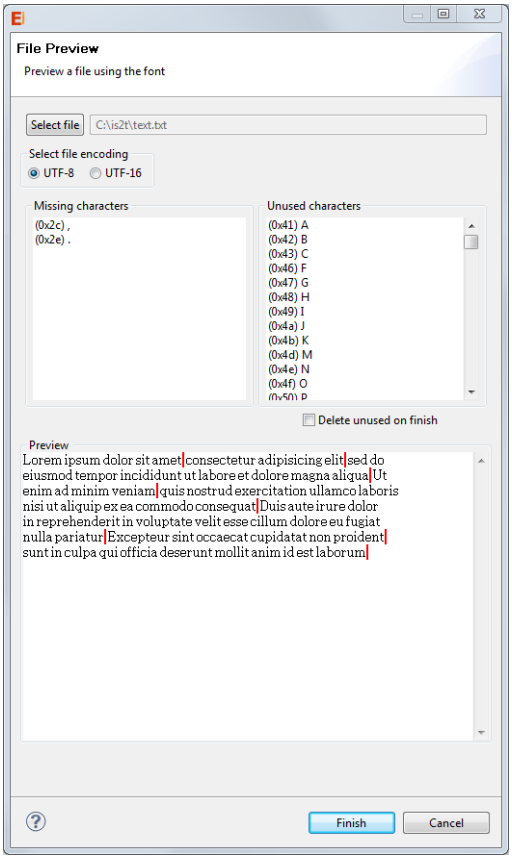


Figure 12.9. Font Preview

12.2.4.5 Removing unused characters

In order to reduce the size of a font file, you can reduce the number of characters in your font to be only those characters used by your application. To do this, create a file which contains all the characters used

by your application (for example, concatenating all your NLS files is a good starting point). Then open the Preview wizard as described above, selecting that file. If you select the check box **Delete unused on finish** then those characters that are in the font but not in the text file will be deleted from the font when you press the **Finish** button, leaving your font containing the minimum number of characters. As this font will only contain characters used by a specific application, it is best to prepare a "complete" font, and then apply this technique to a copy of that font to produce an application specific cut-down version of the font.

12.2.5 Use a MicroEJ Font

A MicroEJ Font must be converted to a format which is specific to the targeted platform. The Font Generator tool performs this operation for all fonts specified in the list of fonts configured in the application launch.

12.2.6 Dependencies

No dependency.

12.2.7 Installation

Font Designer module is already installed in the MicroEJ environment. The module can be installed or not into the platform to let the possibility to the platform user to create or not new fonts.

Note that if the platform user already has a MicroEJ environment which provides the Font Designer module, he/she will able to create a new font even if the platform does not provide the Font Designer module.

In the platform configuration file, check `UI > Font Designer` to install it.

12.2.8 Use

Create a new `ejf` font file or open an existing one to open the Font Designer plugin.

12.3 Font Generator

12.3.1 Introduction

The Font Generator module is an off-board tool that generates fonts ready to be displayed without the need for additional runtime memory. It outputs a raw file for each converted font.

12.3.2 Functional Description

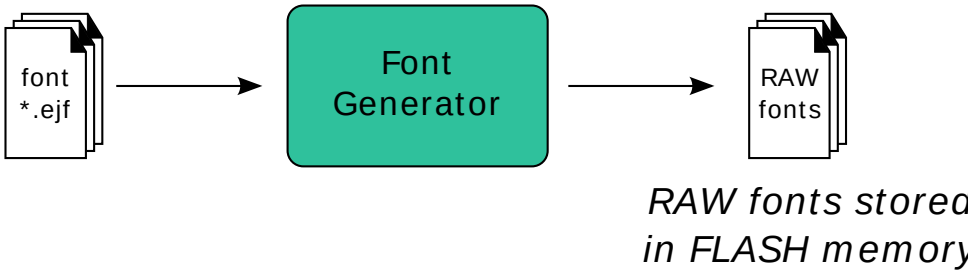


Figure 12.10. Font Generator Principle

Process overview:

1. The user defines, in a text file, the fonts to load.
2. The Font Generator outputs a raw file for each font to convert.
3. The raw files are embedded as (hidden) resources within the Java application. The raw file data is linked into the FLASH memory.

4. When the Java application draws text on the display (or on an image), the font data comes directly from the FLASH memory (the font data is not copied to the RAM memory first).

12.3.3 Pixel Transparency

As mentioned above, each pixel of each character in an `.ejf` file has one of 256 different gray-scale values. However RAW files can have 1, 2, 4 or 8 bits-per-pixel. The required pixel depth is defined in the configuration file. The Font Generator compresses the input pixels to the required depth. In RAW files a pixel value of 0 always represents "fully transparent", and the maximum value (e.g. 3 for 2 bits-per-pixel) represents "fully opaque".

12.3.4 Configuration File

The Font Generator uses a configuration file (called the "list file") for describing fonts that must be processed. The list file is a basic text file where each line describes a font to convert. Each line can have optional parameters which define some ranges of characters to embed in the final raw file, and the required pixel depth. By default all characters available in the input font file are embedded, and the pixel depth is 1 (i.e 1 bit-per-pixel).

Selecting only a specific set of characters to embed reduces the memory footprint. There are two ways to specify a character range: the custom range and the known range. Several ranges can be specified, separated by ";" .

Below is an example of a list file for the Font Generator:

```
myfont
myfont:latin
myfont:latin(5)
```

Figure 12.11. Fonts Configuration File Example

Refer to the chapter “Launch Options” (`Libraries` > `MicroUI` > `Font`) to have more information where specify this fonts configuration file.

12.3.5 Dependencies

- Font Engine Core module (see “Font Engine Core”)

12.3.6 Installation

Font Generator module is an additional tool for MicroUI Java library. When MicroUI module is installed, install this module to be able to embed some additional fonts with the Java application.

If not installed, the platform user will not be able to embed a new font with his/her Java application. He/she will be only able to use the system fonts specified during the MicroUI initialization step (see “Static Initialization”).

In the platform configuration file, check `UI > Font Generator` to install it.

12.3.7 Use

The MicroUI Font APIs are available in the class `ej.microui.io.DisplayFont`.

13 MWT

13.1 Introduction

The MWT module defines a high-level widget framework for embedded devices. For more information please consult the MWT Profile Specification [MWT].

13.2 Widgets

Widgets are visual elements displayed on graphic interfaces. They are a fundamental pieces of graphical application because they are the bridge between user and application/system.

Note that MWT does not provide any widgets, only the possibility to create and use them.

The MWT Widgets specification defines a lightweight set of widgets based on the MWT (Micro Widget Toolkit) framework. They are as small as possible and their default look and feel is made to be quick to render and as visually pleasing as possible.

The widgets library version reference has the form:

```
WIDGETS-x.y-a.b.c
```

and contains two version numbers:

- `x.y`: current widgets library API version.
- `a.b.c`: current I2ST implementation of widgets API.

The MWT widgets can be used as a standard library such as MWT library. However the widgets source is available in the `javaLibs` folder. You can import and modify the default widget behavior, however you will be responsible for the resulting behavior and potential bugs.

13.3 Transition

`Transition` (Flow on MWT) library apply Flow library (see “Flow”) on MWT and using Motion (see “Motion”).

It manages the transition between pages. The motion can be changed as well as the transition.

Some transitions (based on translation) are available:

- **Horizontal.** The pages move horizontally.
- **Vertical.** The pages move vertically.
- **Diagonal.** The pages move diagonally.

The duration of the transition and the period between steps can be configured. The transition way can be reversed. It is possible to choose if:

- the old page do not move and the new page come over for forward transition, the new page goes out for backward transtion (stack-like effect).
- Both pages move, the new one pushing the old one out.

13.4 Dependencies

MWT module requires:

- MicroUI module (see “MicroUI”).

MWT Widgets module requires:

- MWT module.

MWT Flow module requires:

- MWT module.
- Flow module.

13.5 Installation

MWT, MWT Widgets and MWT Flow Java libraries are additional module. In the platform configuration file, check `UI > MWT` , `UI > MWT Widgets` and / or `UI > MWT Flow` to install them.

13.6 Use

Three classpath variables named `MWT-1.0`, `MWT-WIDGETS-1.2-1.1.5` and `EJ.FLOW.MWT-2.0`, are available.

14 Story Board

The Story Board Engine allows you to quickly design a story board for defining your application's user interface. The resulting story board can be run on the simulator and can be also exported to the real hardware. This allows user interaction validation, taking into account details such as touch panel accuracy or physical buttons placement.

The Story Board Engine is composed of:

- The "Story Board Engine Core" module which decodes and shows at application runtime the platform-dependent story board files generated by the "Story Board Generator".
- A "Story Board Designer" module: graphical tool which runs within the MicroEJ workbench used to build and edit story board scenarios; it stores story board files in a platform-independent format.
- A "Story Board Generator" module, for converting story board files from the platform-independent format into a platform-dependent format.

The three elements are complementary: a scenario must be created and edited with the Story Board Designer before being integrated as a resource thanks the Story Board Generator. Finally the Story Board Engine Core uses the generated scenario at runtime.

14.1 Story Board Engine Core

14.1.1 Introduction

The Story Board Engine Core module is an additional module of the MicroUI module (see “MicroUI”). It contains a Java library called "Story Board Player". This library provides some APIs to launch a scenario from the Java application. It takes as entry point the story board files pre-generated by the Story Board Generator.

14.1.2 Functional Description

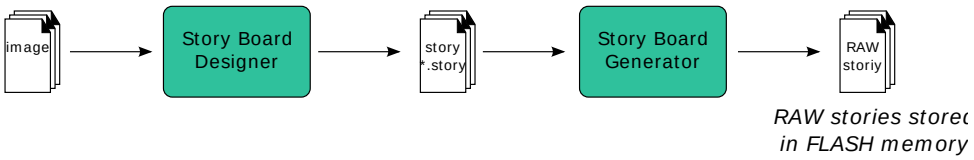


Figure 14.1. Story Board Generation

Process overview:

1. User uses the Story Board Designer module to create a new scenario.
2. Story Board Designer module saves the scenario as a MicroEJ story board file (`*.story` file).
3. The user defines, in a text file, the story board files to load.
4. The Story Board Generator outputs some raw files for each story board to convert (the raw format is display device dependent).
5. The raw files are embedded as (hidden) resources within the Java application. The raw files data are linked into the FLASH memory.
6. When the Java application launches a scenario using the Story Board Player APIs, the Story Board Engine Core decodes the story board raw files. The scenario is now running.

14.1.3 Dependencies

- MicroUI module (see “MicroUI”)

14.1.4 Installation

Story Board Engine Core module is an additional tool for MicroUI Java library. When MicroUI module is installed, install this module to be able to play scenarios.

In the platform configuration file, check `UI > Story Board` to install it.

14.1.5 Use

The Story Board APIs are available in the Java library Story Board Player.

14.2 Story Board Designer

14.2.1 Introduction

The Story Board Designer module is a graphical tool (Eclipse plugin) running within the MicroEJ workbench used to build and edit Story Board scenarios. It stores scenarios in a platform-independent format.

14.2.2 Functional Description

Figure 14.2. Story Board Generation

14.2.3 Create a Storyboard Project

The first step is to create a new MicroEJ project in the MicroEJ workbench. To do this, select the **File** → **New** → **Java Project** item. You will be asked for a project name.

Once the project is created, the next step is creating a storyboard file. Go to **File** → **New** → **MicroEJ Storyboard**. You need to give your file a name, and put it inside the project you just created. Then, you will be asked for the screen resolution you want to use for the storyboard. Set this resolution according to the hardware you are going to use.

The Storyboard Designer will automatically open an editor window to edit your storyboard.

14.2.4 Edit a Storyboard

You can open the Storyboard Designer editor by double clicking on a `.story` file.

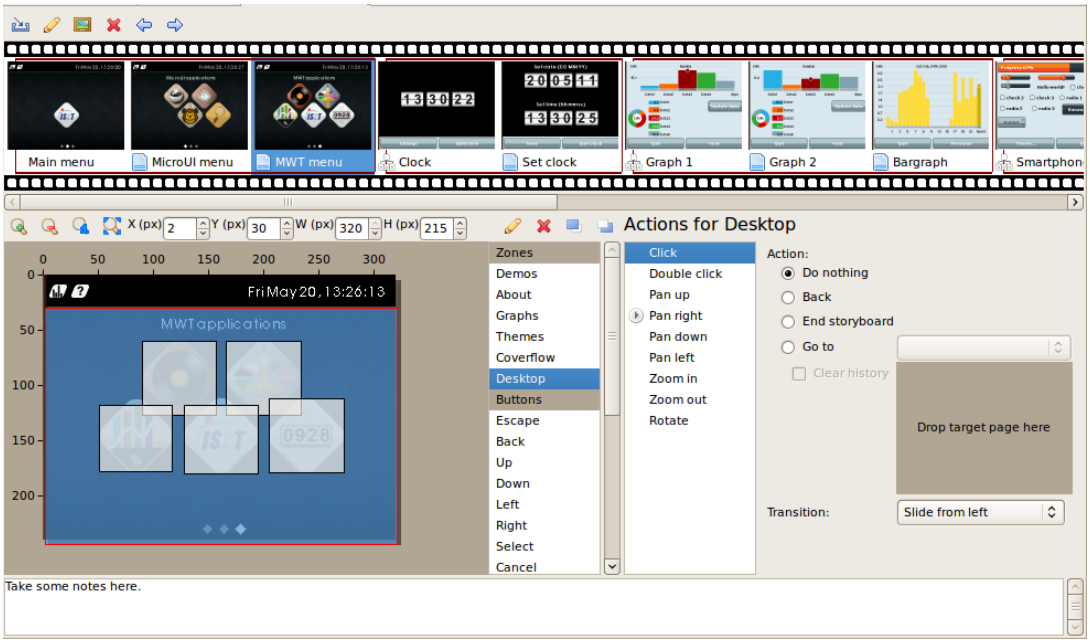


Figure 14.3. Storyboard Designer Application

The interface is made of four main parts:

- The page list at the top shows a preview of all the pages in the storyboard. Pages are associated in groups.
- The zone editor on the left shows the current page, full size. You can define active zones above the widgets in your screen mockup.
- The action editor on the right allows you to define actions to take on occurrences of user interaction.
- The notes area at the bottom is a text field where you can quickly take notes on the current page, for later review.

14.2.4.1 Import and Manage Images

The first step when creating a new Storyboard is to import pages. To do this, click on the **Import** button of the **Page** toolbar. This will open a file selector where you can browse to the image folder and select images. The images are added in the page bar. You can repeat this operation several times to add more images from different folders. Images should be in PNG format and match the screen resolution for best results.

Once your images are in the page list, there are several operations you can do on them.

14.2.4.1.1 Rename Pages

Each page has a name that helps identify it. The default name is the one of the imported file, but you can change it using the **Rename** button, either in the **Page** toolbar or the **Page** pop-up menu.

14.2.4.1.2 Replace a Page Picture

The **Replace picture** button allows you to select a new image for a page. Use this button to update the image, for example if you want to add a new button in the screen mockup. This operation preserves the links with other pages through actions, whereas deleting a page and importing a new one does not.

14.2.4.1.3 Remove a Page

This button allows you to delete a page from the storyboard.

14.2.4.1.4 Sort Pages and Define Groups

Pages can be moved in the list, either by using the **Move left** and **Move right** buttons from the toolbar or by drag and drop in the page list. Besides organizing your pages in a more logical way, this allows you to define groups of page used by the navigation system. Use the **Marker** icon at the bottom of each page to tell if a group should start there.

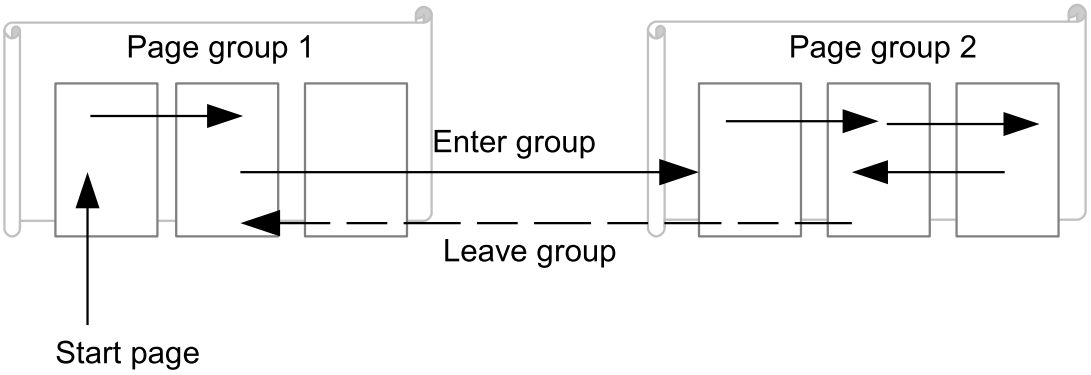


Figure 14.4. Page Groups Navigation Example

The page groups are related to the **Back** action. This action makes the application go out of the current group, back to the page where it was before. The typical use case is a setting window that is reachable from any point in the storyboard, and allows you to get back to the application main page in the state it was before entering the settings.

14.2.4.2 Define Zones

The selected page in the page list is also visible in full size in the zone editor. This editor can be zoomed in and out for easily defining precise zones. Zones define areas of interaction for the touchscreen. To define a zone, just drag the mouse over the zone editor. The zone is shown as a rectangle and added to the zone list.

Once a zone is created, you can move and resize it by clicking on its borders, or using the spinners above the zone editor to define the position and size in pixel units.

Like pages, zones can be renamed and deleted using buttons in the **Zone** toolbar.

14.2.4.2.1 Overlapping Zones

Zones on the screen may overlap. In this case, you can define their relative priorities using the **Raise** and **Lower** buttons in the **Zone** toolbar. When a touch screen event happens, it is first matched to the topmost zone. If the action in this zone is set to **Do Nothing**, then the zone under it is looked at, until there is none left.

An example use is having a zone covering the whole screen and handling pan movements to scroll the screen, while smaller zones above it define clickable items.

14.2.4.2.2 Bind Actions to Zones

Each zone you defined can be associated with actions to trigger. Different gestures are available:

- Click: a single touch on the screen.
- Double click: two clicks in rapid succession.
- Pan: dragging from the zone to the mentioned direction.
- Zoom: multi-touch gesture with two fingers getting away (zoom out) or nearer (zoom in) from each other.

- Rotate: multi-touch gesture with one finger rotating around another.

The default action is to do nothing. This lets other zones under the selected one to handle the action. To define another action, select one of the radio buttons in the action editor.

- **Go to** displays a defined target page. You can set the target either by selecting it in the combo box, or dragging it from the page tool bar and dropping it on the action editor. Dropping a page there will automatically check the **Go to** radio button.
- **Back** gets the application back to the state it was in before entering the current page group.
- **End** terminates the storyboard execution.
- When the selected action is **Go to**, checking the **Clear history** check box will also clear the navigation history until it reaches the requested page or is empty. This is similar to chaining multiple back actions. It can be useful to reach a main desktop screen in an application.

Each action can also have a transition effect. The available effects are **None** which displays the target page immediately and **Slide** which makes it slide on the screen from the given direction.

14.2.4.2.3 Behavior on Exiting the Storyboard

There are two ways of exiting a storyboard, explicitly using the **End** action and using the **Back** action when the navigation history is empty. In both cases, the storyboard player will restore the viewable that was on screen before the storyboard started. If there is no such viewable, an exception is thrown.

14.2.5 Dependencies

No dependency.

14.2.6 Installation

Story Board Designer module is already installed in the MicroEJ environment. The module can be installed or not into the platform to let the possibility to the platform user to create or not new scenarios.

14.2.7 Use

Create a new `story` file or open an existing one to open the Story Board Designer plugin.

14.3 Story Board Generator

14.3.1 Introduction

The Story Board Generator module is an off-board tool that generates scenarios ready to be displayed without the need for additional runtime memory. It outputs a raw files for each converted scenario.

14.3.2 Functional Description

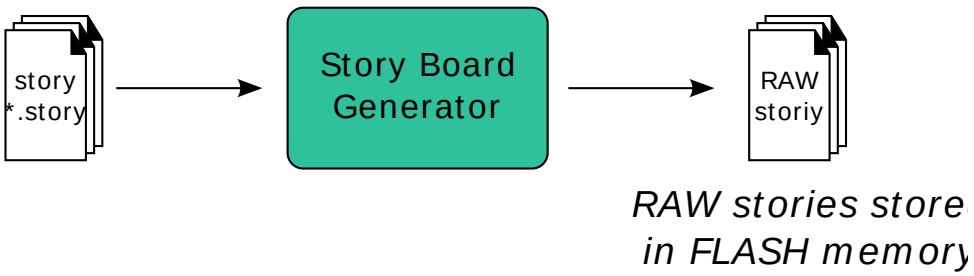


Figure 14.5. Story Board Generator Principle

Process overview:

1. The user defines, in a text file, the scenarios to load.

2. The Story Board Generator outputs some raw files for each scenario to convert.

3. The raw files are embedded as (hidden) resources within the Java application. The raw file data is linked into the FLASH memory.

14.3.3 Configuration File

The Story Board Generator uses a configuration file (called the "list file") for describing scenarios that must be processed. The list file is a basic text file where each line describes a scenario to convert.

Refer to the chapter “Launch Options” (`Libraries` > `MicroUI` > `Story Board`) to have more information where specify this scenarios configuration file.

14.3.4 Dependencies

- Story Board Engine Core module (see “Story Board Engine Core”)

14.3.5 Installation

Story Board Generator module is a part of the Story Board Engine Core module. Install it to be able to convert platform independent scenarios to platform dependent scenarios.

14.3.6 Use

The MicroUI Font APIs are available in the class `ej.microui.io.DisplayFont`.

15 Motion

15.1 Introduction

`Motion` library helps animating application by defining a simple framework and some default motions.

A motion is a numerical value changing over time. It defines starting and stopping positions and a duration. It compute a position at any point in time.

Three types of motions can be used:

- **Ease In.** Starts slowly then speed up.
- **Ease Out.** Starts quickly then slow down.
- **Ease In-Out.** Starts slowly then speed up then slow down.

Provided motions are:

- **Linear.** Represents a uniform linear motion: the velocity is constant (no acceleration or deceleration).
- **Ease.** Represents a motion that accelerate or decelerate over time.
- **Bounce.** Represents a motion that bounce on starting or stopping position.
- **Elastic.** Represents a motion that oscillate over starting or stopping position.

15.2 Dependencies

No dependency.

15.3 Installation

Motion Java libraries ia an additional module. In the platform configuration file, check `UI > Motion` to install it.

15.4 Use

The classpath variable named `EJ.MOTION-2.0`, is available.

16 Flow

16.1 Introduction

`Flow` library helps navigating between the pages of an application.

It manages the creation of pages, the history management.

Three history management policies are available:

- **Lightweight.** Only the page type is stored and it is re-created dynamically when needed. It offers a light memory consumption.
- **Caching.** The page is stored as is. It avoid recreating the page (for example if the creation is long, or if there is enough memory available).
- **Tracking.** The page type and its current state are stored and the is re-created with this state dynamically when needed. It offers a light memory consumption.

16.2 Dependencies

No dependency.

16.3 Installation

Flow Java libraries ia an additional module. In the platform configuration file, check `UI > Flow` to install it.

16.4 Use

The classpath variable named `EJ.FLOW-2.0`, is available.

17 Simulation

17.1 Introduction

A major strength of the MicroEJ environment is that it allows applications to be developed and tested in a simulator rather than on the target device, which may not yet be built. To make this possible for devices that have a display or controls operated by the user, such as a touch screen or buttons, the simulator must connect to a "mock" of the control panel (the "Front Panel") of the device. This mock is called the *mockFP*. The mockFP generates a graphical representation of the required front panel, and is displayed in a window on the user's development machine when the application is executed in the simulator. The mockFP is the equivalent of the five embedded modules (Display, Inputs, LEDs, Alpha Numeric Display and Audio) of the EmbJPF (see “MicroUI”).

The Front Panel mock enhances the Java development environment by allowing User Interface applications to be designed and tested on the computer rather than on the target device (which may not yet be built). The mock interacts with the user's computer in two ways:

- output: LEDs, graphical displays.
- input: buttons, joystick, touch, haptic sensors

17.2 Functional Description

1. Creates a new Front Panel project and associates it with a work in progress JPF.
2. Creates an image of the required front panel. This could be a photograph or a drawing.
3. Defines the content and layout of the front panel by editing an XML file (called an fp file). Full details about the structure and content of fp files can be found in [CM3UIREF].
4. Creates images to animate the operation of the controls (e.g. button down image).
5. Creates *Listeners* that generate the same MicroUI input events as the hardware.
6. Creates a *Display Extension* that configures the simulated display to match the real display.
7. Previews the front panel to check the layout of controls and the events they create, etc.
8. Exports the Front Panel project into the work-in-progress JPF.

17.3 The Front Panel Project

17.3.1 Creating a Front Panel Project

A Front Panel project is created using the New Front Panel Project wizard. Select

New → Project... → MicroEJ → Front Panel Project

The wizard will appear:

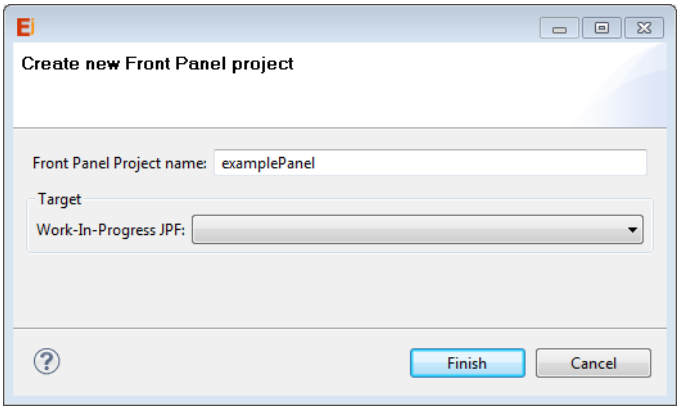


Figure 17.1. New Front Panel Project Wizard

Enter the name for the new project and select the work-in-progress JPF with which this project should be associated. Note that this association can be changed using the **Associated JPF** page of the project's properties.

17.3.2 Project Content

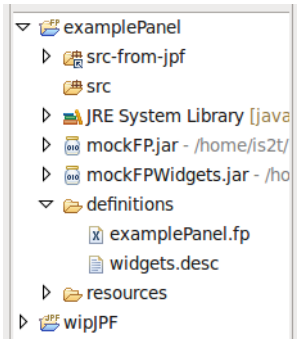


Figure 17.2. Project Content

Note that the project icon has the letters “FP” in the top right corner.

A Front Panel project has the following structure and content:

- The `src-from-jpf` folder is linked to the work-in-progress JPF and contains an interface `com.is2t.microej.microui.Constants` which defines constants mapping event generator names to numeric values. These constants are used when defining `Listeners`, and match the MicroUI definition for the work-in-progress JPF. This interface must not be edited. See below for more details about defining listeners.

- The `src` folder is provided for the definition of `Listeners` and `DisplayExtensions`. It is initially empty. The creation of `Listeners` and `DisplayExtensions` will be explained later.

- The `JRE System Library` is referenced, because a Front Panel project needs to support the writing of Java for the `Listeners` and `DisplayExtensions`.

- The `mockFPWidgets.jar` references the work-in-progress JPF and contains the code for the front panel simulation, the widgets it supports and the types needed to implement `Listeners` and `DisplayExtensions`.

- The `definitions` folder holds the file or files that define the content and layout of the front panel, with a `.fp` extension (the fp file or files), plus some supporting files. A newly created project will

have a single fp file with the same name as the project, as shown above. The content of fp files is explained later in this document.

- The `widgets.desc` file contains descriptions of the widgets supplied with the XPF. It is used by the Front Panel Designer tool and must not be edited.
- The `resources` folder holds images used to create the mockFP. It is initially empty.

17.4 FP File

17.4.1 File Content

The mock engine takes an XML file, the fp file, as input. It describes the panel using mock-widgets: they simulate the drivers, sensors & actuators of the real device. The mock engine generates the graphical representation of the real device, and is displayed in a window on the user's development machine when the application is executed in the simulator.

The following example file describes a typical board with one LCD, a touch panel, three buttons, a joystick and four LEDs:

```
<?xml version="1.0"?>
<frontpanel
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns="http://xml.is2t.com/ns/1.0/frontpanel"
 xsi:schemaLocation="http://xml.is2t.com/ns/1.0/frontpanel .fp1.0.xsd">

 <description file="widgets.desc"/>

 <device name="MyBoard" skin="myboard.png">
  <body>
   <pixelatedDisplay id="0" x="162" y="114" width="320" height="240"
 initialColor="0x000000" extensionClass="com.is2t.microej.fp.DisplayExtension"/>
   <pointer id="0" x="162" y="114" width="320" height="240" touch="true"
 listenerClass="com.is2t.microej.fp.PointerListenerImpl"/>

   <led2states id="0" x="277" y="374" ledOff="led0_0.png" ledOn="led0_1.png"
 overlay="false"/>
   <led2states id="1" x="265" y="374" ledOff="led1_0.png" ledOn="led1_1.png"
 overlay="false"/>
   <led2states id="2" x="254" y="374" ledOff="led2_0.png" ledOn="led2_1.png"
 overlay="false"/>
   <led2states id="3" x="242" y="372" ledOff="led3_0.png" ledOn="led3_1.png"
 overlay="false"/>

   <repeatPush id="0" x="250" y="395" skin="Button1_0.png" pushedSkin="Button1_1.png"
 repeatPeriod="250" listenerClass="com.is2t.microej.fp.ButtonListener"/>
   <repeatPush id="1" x="322" y="395" skin="Button1_0.png" pushedSkin="Button1_1.png"
 repeatPeriod="250" listenerClass="com.is2t.microej.fp.ButtonListener"/>
   <repeatPush id="2" x="456" y="395" skin="Button1_0.png" pushedSkin="Button1_1.png"
 repeatPeriod="250" listenerClass="com.is2t.microej.fp.ButtonListener"/>
   <joystick id="0" x="368" y="375" skin="Joy0.png"
 mask="JoyMask.png" enterSkin="Joy1.png" upSkin="Joy_UP.png"
 downSkin="Joy_DOWN.png" leftSkin="Joy_LEFT.png" rightSkin="Joy_RIGHT.png"
 listenerClass="com.is2t.microej.fp.JoystickListenerImpl"/>
  </body>
 </device>
</frontpanel>
```

The `description` element must appear exactly as shown. It refers to the `widgets.desc` file mentioned above.

The `device skin` must refer to a `png` file in the `resources` folder. This image is used to render the background of the front panel. The widgets are drawn on top of this background.

The `body` element contains the elements that define the widgets that make up the front panel. The name of the widget element defines the type of widget. The set of valid types is determined by the work-in-

progress JPF associated with the project. Every widget element defines an `id`, which must be unique for widgets of this type, and the `x` and `y` coordinates of the position of the widget within the front panel (0,0 is top left). There may be other attributes depending on the type of the widget.

Refer to the UI Reference Manual for more information about the content of the fp file.

The file and tags specifications are available in [CM3UIREF].

17.4.2 Working with fp Files

To edit an fp file, open it using the Eclipse XML editor (right-click on the fp file, select **Open With** → **XML Editor**). This editor features syntax highlighting and checking, and content-assist based on the schema (XSD file) referenced in the fp file. This schema is a hidden file within the project's definitions folder. An incremental builder checks the content of the fp file each time it is saved and highlights problems in the **Eclipse Problems** view, and with markers on the fp file itself.

A preview of the front panel can be obtained by opening the Front Panel Preview (either **Window** → **Show View** → **Other...** → **MicroEJ** → **Front Panel Preview** or **Window** → **Open Perspective** → **Other...** → **MicroEJ** followed by **Window** → **Reset Perspective...**).

The preview updates each time the fp file is saved.

A typical working layout is shown below.

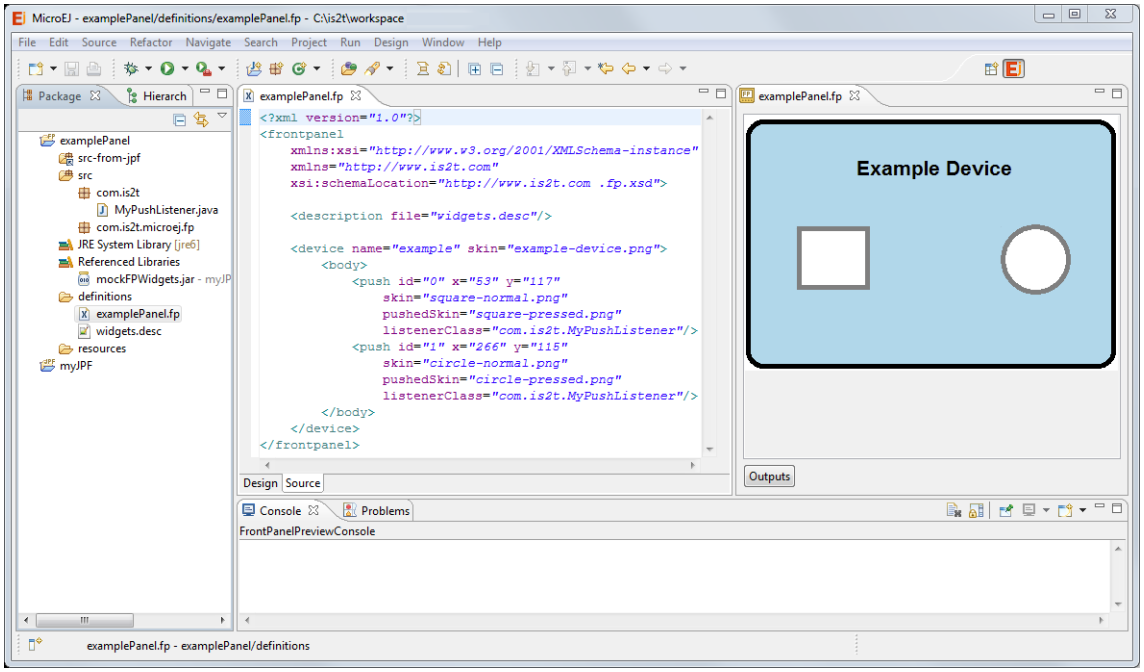


Figure 17.3. Working Layout Example

Within the XML editor, content-assist is obtained by pressing **ctrl**+**space**. The editor will list all the elements valid at the cursor position, and insert a template for the selected element.

17.4.3 Skins and Filters

The widgets which simulate the input devices use images (or "skins") to show their current states (pressed and released). The user can change the state of the widget by clicking anywhere on the skin: it is the active area. This active area is, by default, rectangular.

These skins can be associated with an additional image called a `filter` or `mask`. This image defines the widget's active area. It is useful when the widget is not rectangular.

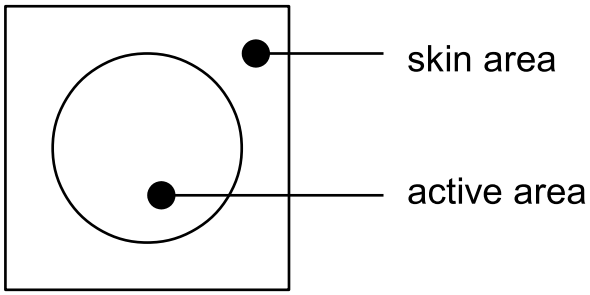


Figure 17.4. Active Area

The filter image must have the same size as the skin image. The active area is delimited by the color 0xFF00FF (pink). Every pixel in the filter image which is not this color is considered not part of the active area.

17.5 Extensions

The mock engine also requires several extensions to specify the display characteristics and how to react to input events. These extensions names (package and classname) are specified in the fp file.

17.5.1 Display

In order to use a display, a `Display` widget needs to be specified, along with a `DisplayExtension` which defines the characteristics of the display.

Here is an example snippet of a `.fp` file for specifying a `Display` widget:

```
<pixelatedDisplay
    id="0"
    x="162" y="114"
    width="320" height="240"
    initialColor="0x000000"
    extensionClass="com.is2t.MyDisplayExtension"/>
```

Figure 17.5. `.fp` file - display example

The extension class attribute specifies the fully qualified class name of a class that extends an abstract subclass of `com.is2t.microej.frontpanel.display.DisplayExtension`. There are two kinds of extensions:

- Generic extension: the display device pixels memory representation is standard (corresponds to a generic display stack: SWITCH, COPY or DIRECT).
- Custom extension: the display device pixels memory representation is specific (corresponds to the custom display stack).

17.5.1.1 Generic Extension

17.5.1.1.1 Principle

The generic extension implementation must define several display device characteristics:

- the number of bits-per-pixel
- the memory layout
- the pixel color conversion
- etc.

Notes:

- only 16 bits-per-pixel displays without alpha blending, or with a basic alpha blending (fully opaque and fully transparent), are supported by the generic extensions.

17.5.1.1.2 API

The abstract class `GenericDisplay16XYExtension` targets a 16 bits-per-pixel display with the memory layout "line by line" (also called XY), whereas the abstract class `GenericDisplay16YXExtension` targets a 16 bits-per-pixel display with the memory layout "column by column" (also called YX). Similar pairs of classes are provided for the 1, 2 and 4 bits-per-pixel display stacks.

`convertRGBColorToDisplayColor` method is used to convert a 24-bit RGB color into the display pixel memory representation.

`convertDisplayColorToRGBColor` method is used to convert a color in into the display pixel memory representation into a 24-bit RGB color.

`numberOfColors` method returns the number of colors the display can show.

`isColor` method returns true when the display is not a grayscaled display.

`isDoubleBuffered` method returns true when the display uses the double buffering method to render its content.

`getAlphaLevel` method returns the alpha blending level (0/1 means no alpha blending support; 2 means support for fully opaque or fully transparent pixels; no other values are supported).

17.5.1.2 Custom Extension

17.5.1.2.1 Principle

In addition to the same concepts as the generic extension, the custom extension implementation also has the responsibility to decode static images. The static image raw encoding is image generator extension dependent.

17.5.1.2.2 API

The abstract class to implement is `CustomDisplayExtension`.

The methods `convertRGBColorToDisplayColor`, `convertDisplayColorToRGBColor`, `numberOfColors`, `isColor`, `isDoubleBuffered` and `getAlphaLevel` have the same behavior as the generic extension.

`getBPP` method returns the number of bits-per-pixel.

`decodeImmutableImage` method has to decode the image for the given byte array. The decoder has to draw each pixel in the given output image `CustomImage`.

17.5.2 Inputs

17.5.2.1 Widgets and Listeners

17.5.2.1.1 Principle

For widgets that generate events, a `Listener` class must be specified within the `.fp` file.

As an example, consider this snippet of a `.fp` file for defining a push button:

```
<push id="0" x="54" y="117"
    skin="square-normal.png"
    pushedSkin="square-pressed.png"
    listenerClass="com.is2t.MyPushListener" />
```

Figure 17.6. `.fp` file - push example

The value of the `listenerClass` attribute is the fully qualified name of a class which has to implement the `com.is2t.microej.frontpanel.input.listener.PushButtonListener` interface. This class, `com.is2t.MyPushListener`, is written by the developer and implements the `PushButtonListener` interface.

17.5.2.1.2 API

The package `com.is2t.microej.frontpanel.input.listener` provides `Listeners` required by other widgets too:

- `push` requires `PushButtonListener`
- `repeatPush` requires `RepeatButtonListener`
- `joystick` requires `JoystickListener`
- `pointer` requires `PointerListener`

A listener implementation can target several widgets. Each action method receives the ID of the widget that has changed as a parameter.

17.5.2.2 Event Generator

17.5.2.2.1 Principle

The `Listener` needs to be written to generate the same events that the hardware will. In order to send an event, the implementation of a `Listener` needs to use the `EventGenerator` class provided. For example, a `PushButtonListener` can generate button events by calling static methods `sendButtons[...]Event`.

Note that a `Listener` can generate events using any methods of the provided `EventGenerator`. In some cases, this may provide useful functionality.

Each `EventGenerator` method requires an unique ID of the MicroUI event generator it targets. This unique ID is available in the interface `com.is2t.microej.microui.Constants` which has been generated by the Static MicroUI Initializer tool.

17.5.2.2.2 API

The `EventGenerator` class targets six MicroUI event generators:

- `EventGenerator` (IS2T generic event generator): `sendEvent`, `sendEvents`
- `CommandEventGenerator`: `sendCommandEvent`
- `ButtonsEventGenerator`: `sendButtons[...]Event`
- `PointerEventGenerator`: `sendPointer[...]Event`
- `TouchEventGenerator` (IS2T touch event generator): `sendTouch[...]Event`
- `StatesEventGenerator`: `sendState[...]Event`

17.6 Dependencies

- MicroUI module (see “MicroUI”)
- Display module (see “Display”): this module gives the characteristics of the graphical display useful to configure the Front Panel.

17.7 Installation

Front Panel is an additional module for MicroUI Java library. When MicroUI module is installed, install this module to be able to simulate UI drawings on the simulator.

In the platform configuration file, check `UI > Front Panel` to install it. When checked, the properties file `frontpanel > frontpanel .properties` is required during platform creation to configure the module. This configuration step is used to identify and configure the front panel.

The properties file have to / can contain the following properties:

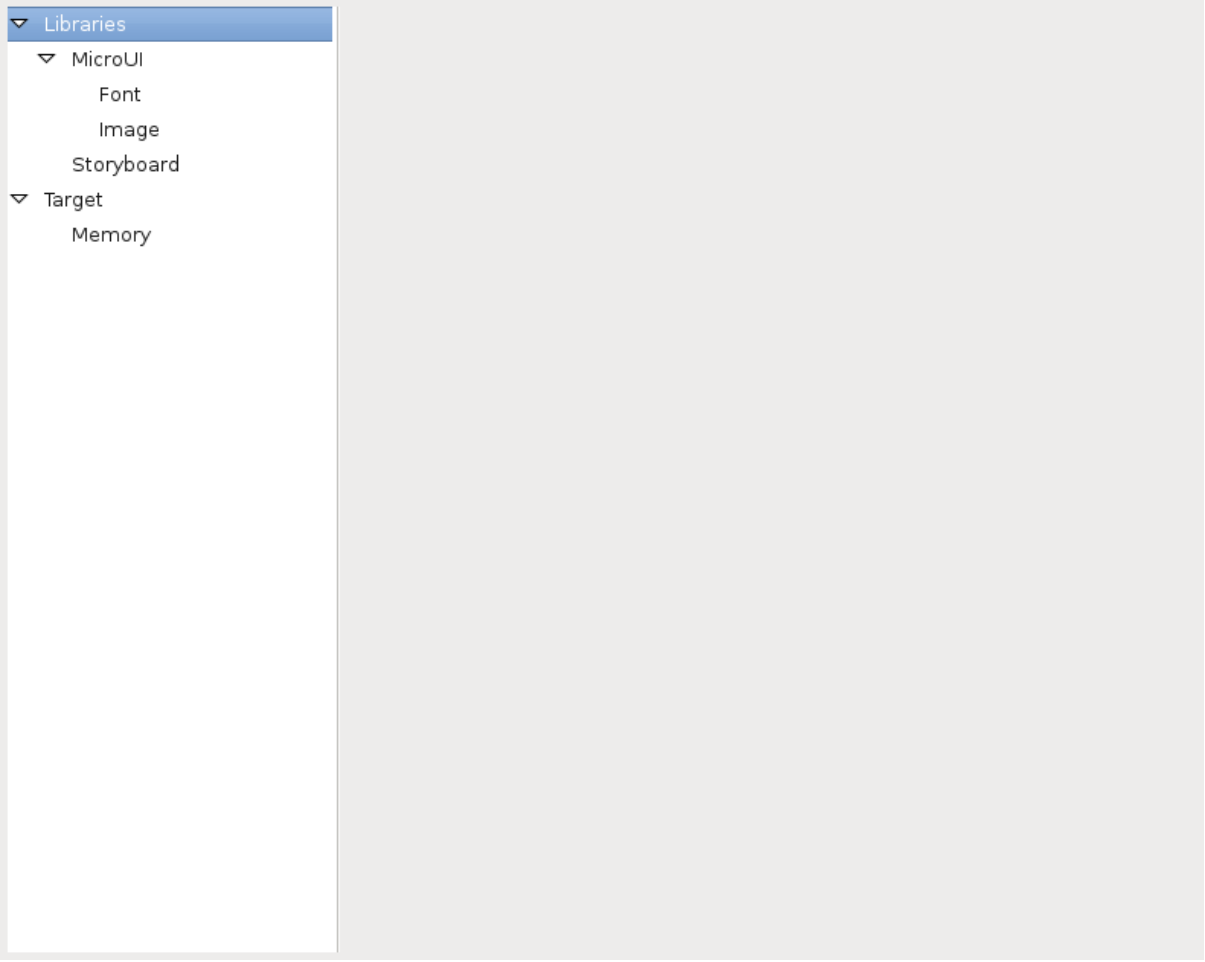
- `project.name` [mandatory]: Defines the name front panel project (same workspace as the platform configuration project). If not exists, a new project will be created.
- `fpFile.name` [optional, default value is "" (*empty*)]: Defines the front panel file (*.fp) to export (in case of "project.name" contains several fp files). If empty or unspecified, first ".fp" file found will be exported.
- `decoder.png.enabled` [optional, default value is "true"]: Defines either the runtime PNG decoder have to be enabled.
- `decoder.bmpm.enabled` [optional, default value is "true"]: Defines either the runtime BMP Mono-chrom decoder have to be enabled.

17.8 Use

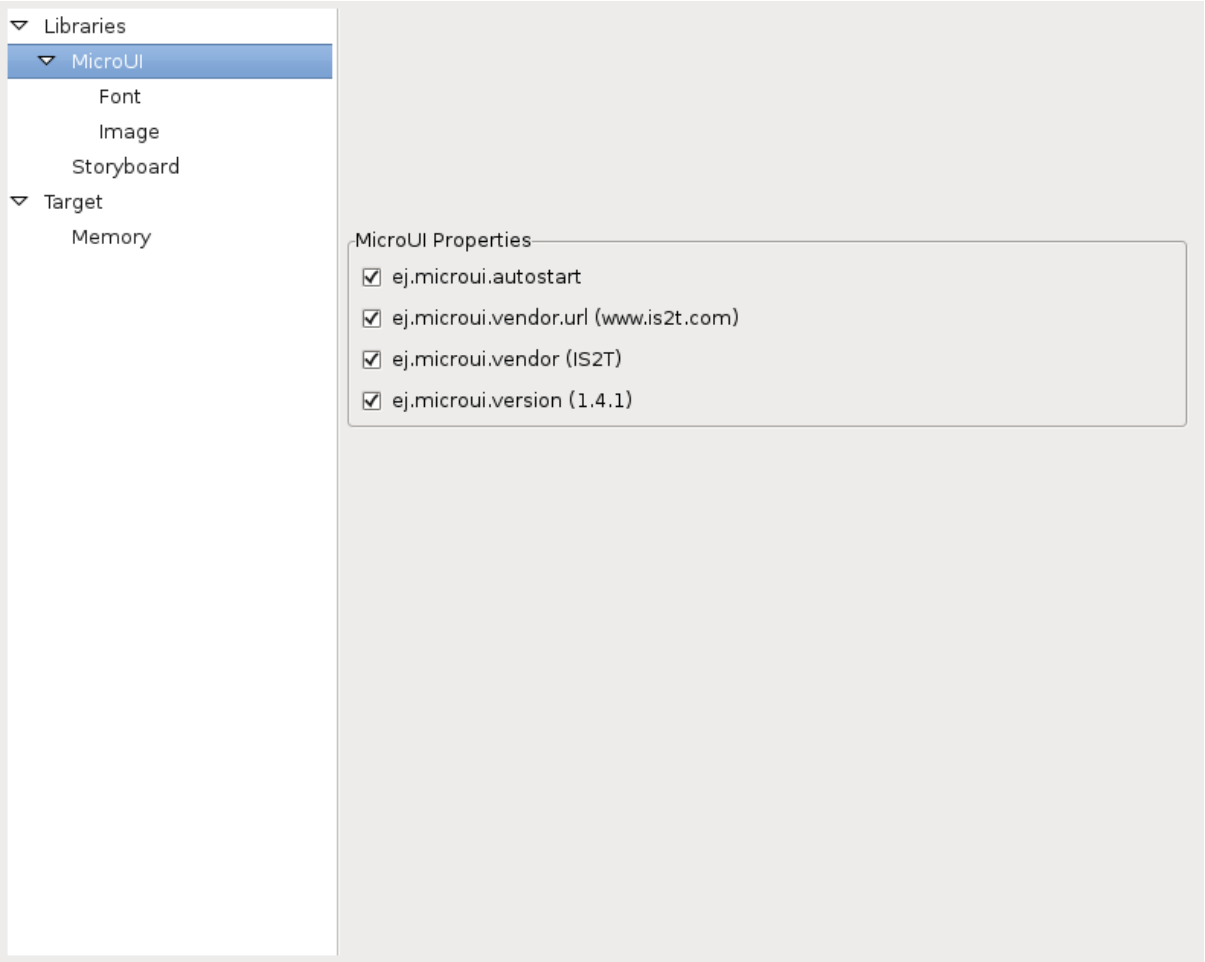
Launch a MicroUI application on the simulator to run the Front Panel.

18 Launch Options

18.1 Category: Libraries



18.1.1 Category: MicroUI



18.1.1.1 Group: MicroUI Properties

18.1.1.1.1 Option(checkbox): ej.microui.autostart

Default value: `checked`

Description:

Check this option to start automatically MicroUI framework before the user's main method (`MUI_01`). If it is not checked, the application has to call `MicroUI.start()` method to start the MicroUI framework.

18.1.1.1.2 Option(checkbox): ej.microui.vendor.url (www.is2t.com)

Default value: `checked`

Description:

Adds the Java property `ej.microui.vendor.url`.

18.1.1.1.3 Option(checkbox): ej.microui.vendor (IS2T)

Default value: `checked`

Description:

Adds the Java property `ej.microui.vendor`.

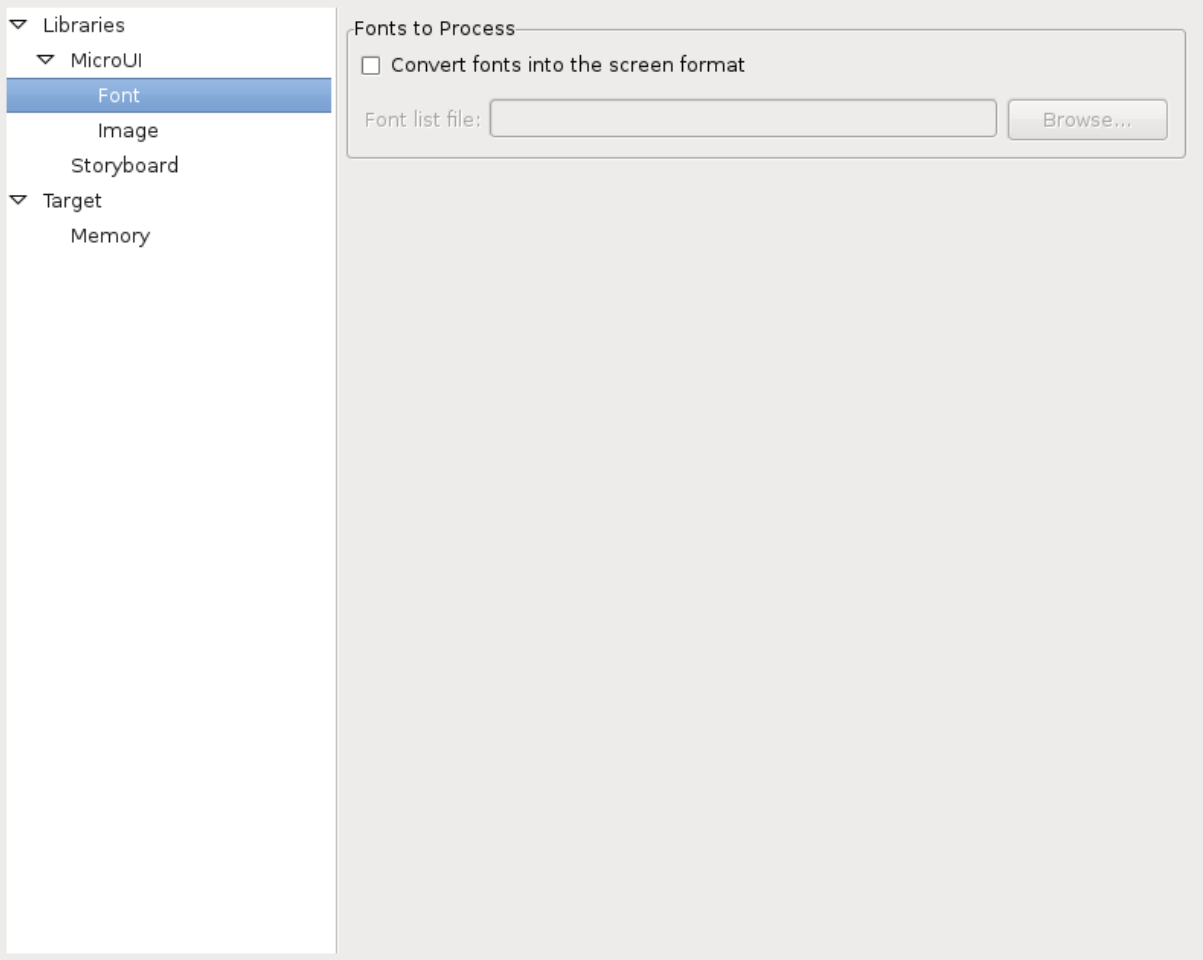
18.1.1.1.4 Option(checkbox): ej.microui.version (1.4.1)

Default value: `checked`

Description:

Adds the Java property `ej.microui.version`.

18.1.1.2 Category: Font



18.1.1.2.1 Group: Fonts to Process

Description:

This group allows to select a file describing the font files which need to be converted into the EmbJPF format. At MicroUI runtime, the pre-generated fonts will be read from the flash memory without any modifications (see MicroUI specification).

18.1.1.2.1.1 Option(checkbox): Convert fonts into the screen format

Default value: `unchecked`

Description:

When checked, enables the next option `Fonts list` file. When the next option is disabled, there is no check on the file path validity.

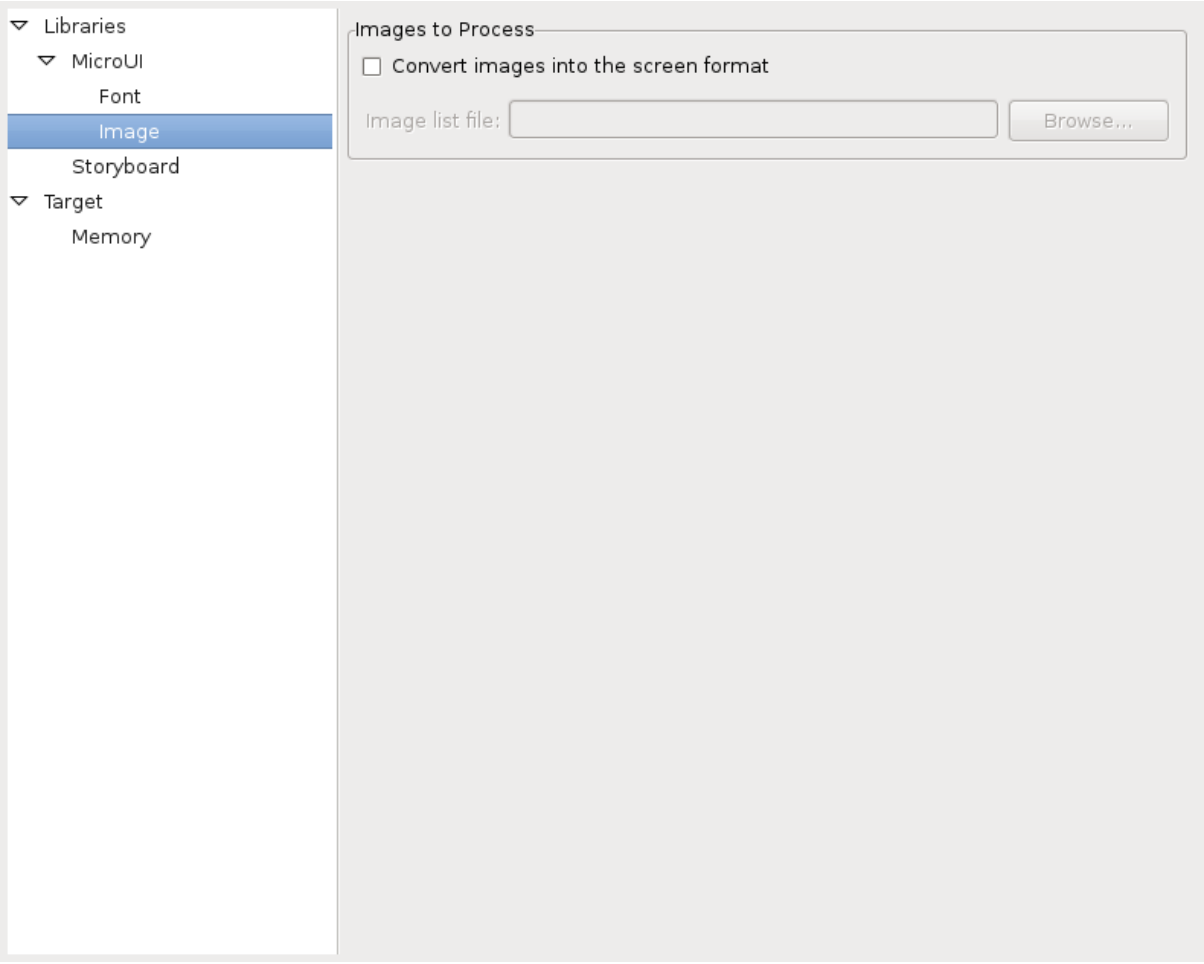
18.1.1.2.1.2 Option(browse): Font list file

Default value: `(empty)`

Description:

Browse to select a font list file. Refer to Font Generator chapter for more information about the font list file format.

18.1.1.3 Category: Image



18.1.1.3.1 Group: Images to Process

Description:

This group allows to select a file describing the image files which need to be converted into the EmbJPF format. At MicroUI runtime, the pre-generated images will be read from the flash memory without any modifications (see MicroUI specification).

18.1.1.3.1.1 Option(checkbox): Convert images into the screen format

Default value: `unchecked`

Description:

When checked, enables the next option `Images list` file. When the next option is disabled, there is no check on the file path validity.

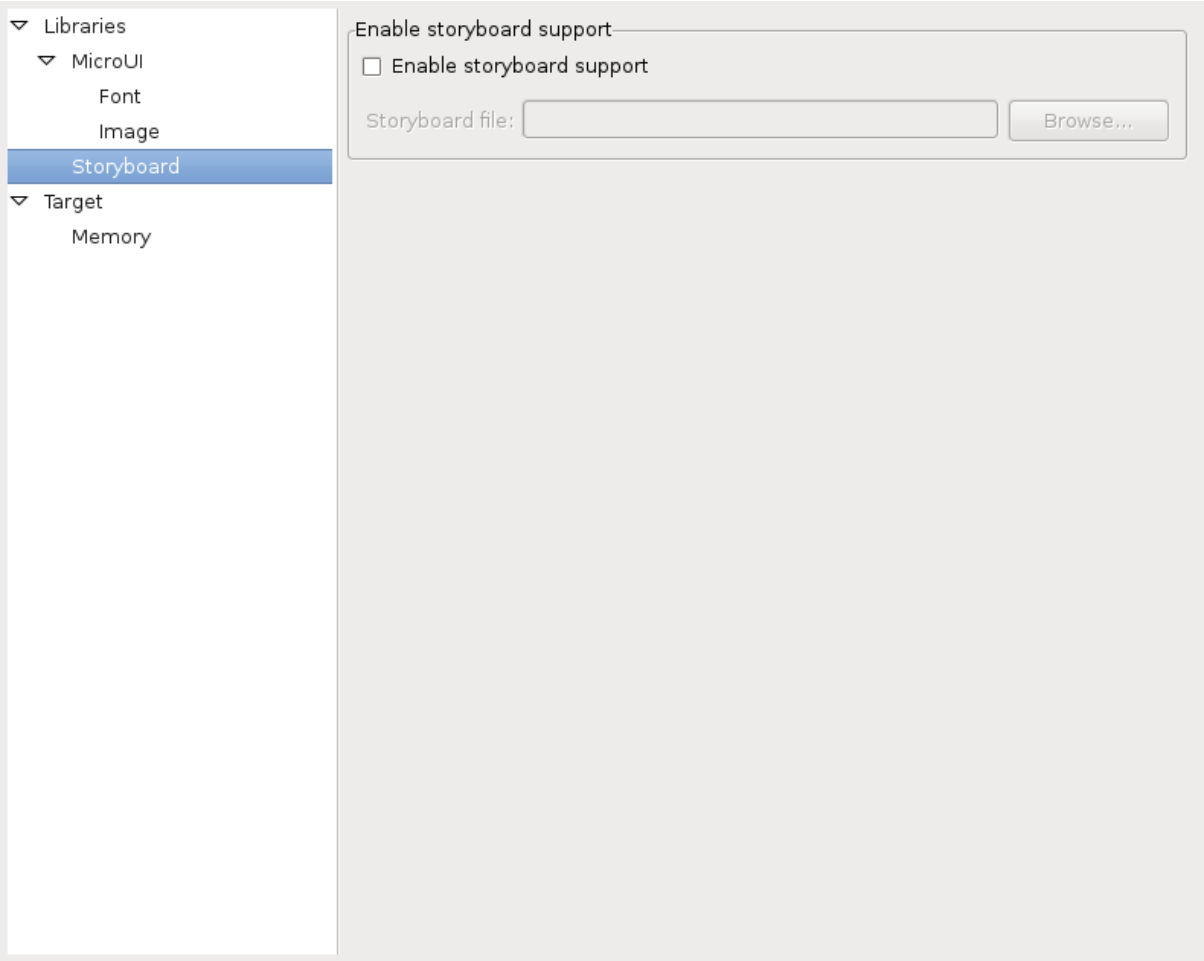
18.1.1.3.1.2 Option(browse): Image list file

Default value: `(empty)`

Description:

Browse to select an image list file. Refer to Image Generator chapter for more information about the image list file format.

18.1.2 Category: Storyboard



18.1.2.1 Group: Enable storyboard support

Description:

Options related to the Storyboard designer.

18.1.2.1.1 Option(checkbox): Enable storyboard support

Default value: `unchecked`

Description:

Enable storyboard data generation.

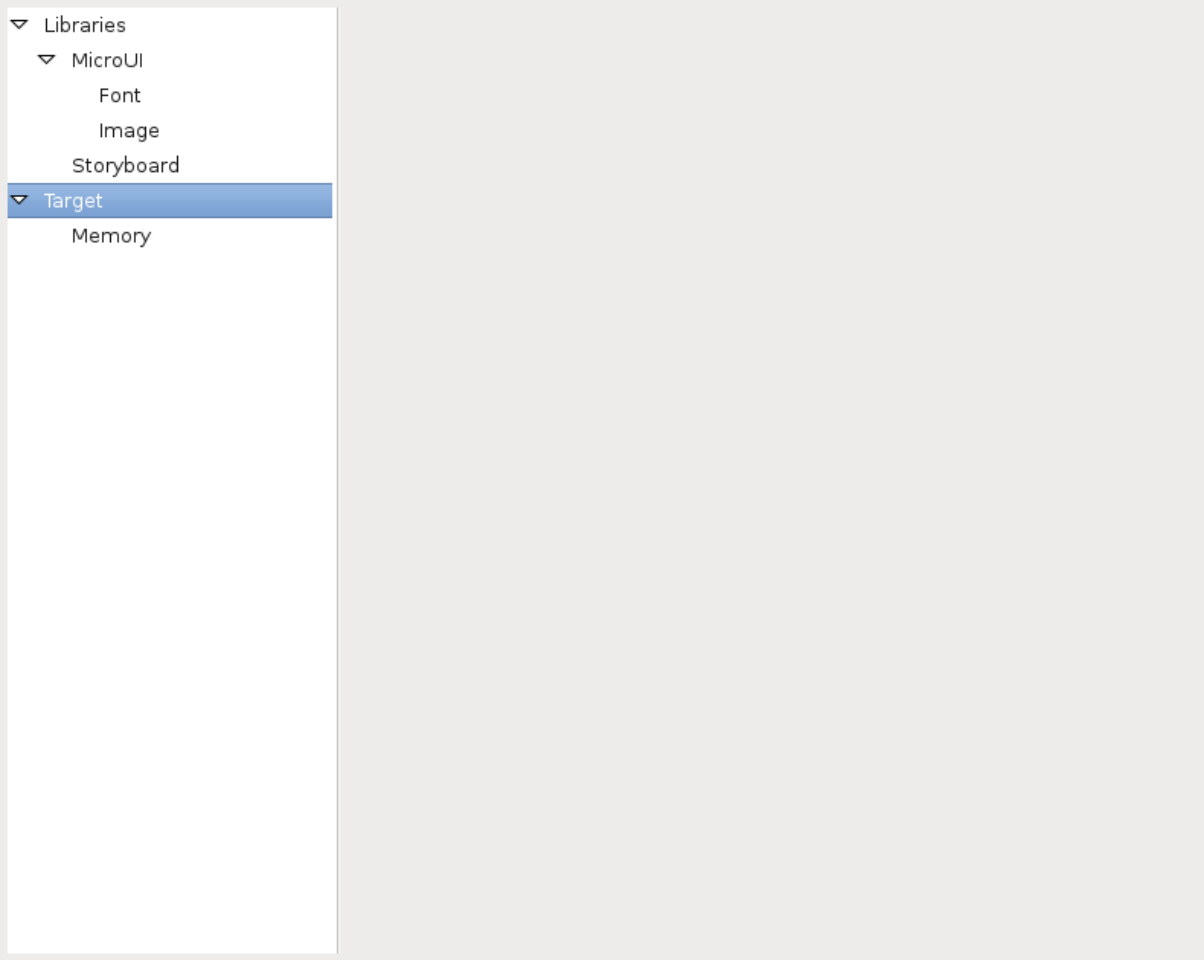
18.1.2.1.2 Option(browse): Storyboard file

Default value: `(empty)`

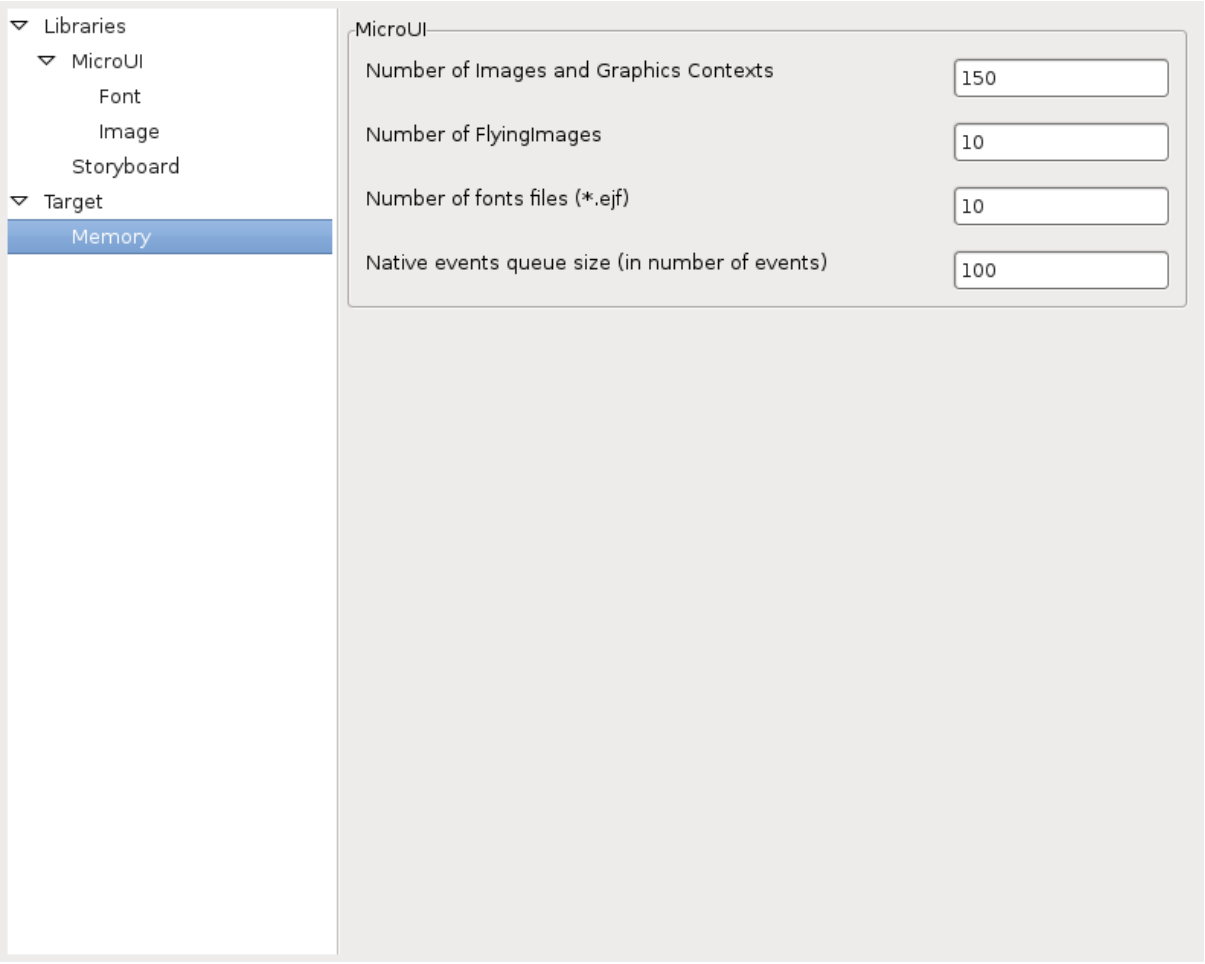
Description:

Browse to load a storyboard file. Refer to Storyboard Designer chapter in MicroEJ GUI Pack documentation to have more information about storyboard file format.

18.2 Category: Target



18.2.1 Category: Memory



18.2.1.1 Group: MicroUI

18.2.1.1.1 Option(text): Number of Images and Graphics Contexts

Default value: `150`

Description:

Specifies the number of Image the application can open at the same time. If the limit is reached at runtime an OutOfMemory error is thrown.

18.2.1.1.2 Option(text): Number of FlyingImages

Default value: `10`

Description:

Specifies the number of Flying Image the application can open at the same time. If the limit is reached at runtime an OutOfMemory error is thrown.

18.2.1.1.3 Option(text): Number of fonts files (*.ejf)

Default value: `10`

Description:

Specifies the number of fonts files the platform can load. If the number of fonts files to load is higher than the specified number, a warning is showed and the last fonts files are not loaded.

18.2.1.1.4 Option(text): Native events queue size (in number of events)

Default value: `100`

Description:

Specifies the size of the native events queue.

19 Document History

Date	Revision	Description
January 11th 2013	A	First release
July 11th 2013	B	Update for STM32Java 2.0.0
June 19th 2014	C	Update for STM32Java 3.0.0